UNIVERSIDADE FEDERAL DO CEARÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

Otávio Alcântara de Lima Júnior

SKMOTES: UM KERNEL SEMIPREEMPTIVO PARA NÓS DE REDES DE SENSORES SEM FIO

FORTALEZA - CEARÁ
OUTUBRO - 2011

Otávio Alcântara de Lima Júnior

SKMOTES: UM KERNEL SEMIPREEMPTIVO PARA NÓS DE REDES DE SENSORES SEM FIO

DISSERTAÇÃO

Dissertação submetida ao corpo docente da Coordenação do Programa de Pós-Graduação em Engenharia de Teleinformática da **Universidade Federal do Ceará** como parte dos requisitos necessários para obtenção do grau de MESTRE EM ENGENHARIA DE TELEINFORMÁTICA.

Área de concentração: Sinais e Sistemas

Prof. Dr. Paulo Cesar Cortez
(Orientador)

Prof. Dr. Helano de Sousa Castro
(Co-orientador)

FORTALEZA - CEARÁ
2011

OTÁVIO ALCÂNTARA DE LIMA JÚNIOR

SKMOTES: UM KERNEL SEMI-PREEMPTIVO PARA NÓS DE REDES DE SENSORES SEM FIO

Dissertação submetida à Coordenação do Programa de Pós-Graduação em Engenharia de Teleinformática, da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Engenharia de Teleinformática.
Área de concentração: Sinais e Sistemas.

Aprovada em 07/10/2011.

BANCA EXAMINADORA

Prof. Dr. Paulo César Cortez (Orientador)
Universidade Federal do Ceará - UFC

Prof. Dr. Hélano de Sousa Castro
Universidade Federal do Ceará - UFC

Prof. Dr. Giovanni Cordeiro Barroso
Universidade Federal do Ceará - UFC

Prof. Dr. Paulo Eigi Miyagi
Universidade Estadual de Sao Paulo - USP

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Ciências e Tecnologia

L699s Lima Júnior, Otávio Alcântara de.
SKMotes : um kernel semipreemptivo para nós de redes de sensores sem fio / Otávio Alcântara de Lima Júnior. – 2011.
91 f. : il., enc. ; 30 cm.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Tecnologia, Departamento de Engenharia de Teleinformática, Programa de Pós-Graduação em Engenharia de Teleinformática, Fortaleza, 2011.
Área de Concentração: Sinais e Sistemas.
Orientação: Prof. Dr. Paulo César Cortez.
Coorientação: Prof. Dr. Helano Sousa Castro.

1. Sistemas operacionais. 2. Redes sensoriais. 3. Algoritmos. 4. Sistemas de controle inteligente. I. Título.

CDD 621.38

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

ADC	Conversor Analógico-Digital
CI	Circuito Integrado
CH	Cluster Head
CPU	Central Processing Unit
EEABR	Energy-Efficient Ant-based Routing Algorithm
E/S	Entrada e Saída
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GPIO	General Purporse Input and Output
GPS	Sistema de Posicionamento Global
I2C	Inter-Intergrated Circuit
PC	Personal Computer
PLD	Programmable Logic Device
MCU	Microcontrolador
RAM	Random Access Memory
RF	Rádio Frequência
RSSF	Rede de Sensor sem Fio
RX	Receiver
SKMotes	Small Kernel for Motes
SO	Sistema Operacional
SOC	System On Chip
SPI	Serial Peripheral Interface
TX	Transmitter
UART	Universal Assincronous Receiver Transmitter

VHDL VHSIC Hardware Description Language

Resumo

Redes de Sensores sem Fio (RSSFs) são fruto dos recentes avanços nas tecnologias de sistemas micro-eletro-mecânicos, circuitos integrados de baixa potência e comunicação sem fio de baixa potência. Estes avanços permitiram a criação de minúsculos dispositivos computacionais de baixo custo e baixa potência, capazes de monitorar grandezas físicas do ambiente e estabelecer comunicação uns com os outros. Estes dispositivos, denominados nós sensores, são dotados de um microcontrolador simples, elementos sensores, rádio transceptor e fonte de alimentação. Desenvolver aplicações para RSSFs é um grande desafio. O sistema operacional (SO) é um componente essencial de um projeto de uma aplicação para RSSFs. Em relação ao modelo de concorrência, podem-se dividir os SOs em duas categorias: baseados em eventos e baseados em *threads*. O modelo baseado em eventos cria dificuldades ao programador para controlar os fluxos de execução e não se ajusta a problemas com longos períodos de computação. Por outro lado, o modelo baseado em *threads* tem alto consumo de memória, mas fornece um modelo de programação mais simples e com bons tempos de resposta. Dentro desse contexto, esta dissertação propõe um novo SO para RSSFs, chamado SKMotes, que explora as facilidades de programação do modelo *threads* aliadas à baixa ocupação de memória. Este SO utiliza um modelo de concorrência baseado em *threads*, mas não completamente preemptivo, pois em dado momento apenas um subconjunto das *threads* do sistema está executando no modo preemptivo baseado em prioridades. O restante das *threads* permanece em espera, ocupando apenas um contexto mínimo de execução, que não contempla a pilha de dados. O principal objetivo desse modelo é prover tempos de resposta baixos para *threads* de alta prioridade, ao mesmo tempo que garante baixo consumo de energia e ocupação de memória mais baixa do que *kernels* preemptivos. Estas características permitem que o SKMotes seja empregado em aplicações de RSSFs que utilizem um conjunto de tarefas orientadas à E/S e a longos períodos de computação. Por exemplo, aplicações de RSSFs que realizem

funções de compressão de dados, criptografia, dentre outras. A avaliação de desempenho do SO proposto foi realizada em um ambiente de testes, baseado em uma FPGA, projetado para esta dissertação, que permite realizar medições da utilização da CPU e do tempo de resposta das *threads*, ao mesmo tempo em que interage com a plataforma do nó sensor através da interface de comunicação serial. Este ambiente de testes pode ser reutilizado em diferentes cenários de avaliação de desempenho de sistemas computacionais baseados em microcontroladores. Os testes de avaliação de desempenho mostram que, para os casos de teste realizados, o SKMotes apresenta ocupação do processador equivalente às soluções baseadas em *multithreading* preemptivo, mas com consumo de memória de dados, em média, 20% menor. Além disso, o SKMotes é capaz de garantir tempos de respostas, em média, 34% inferiores às soluções baseadas em *kernels* de eventos. Quando se avalia apenas os casos de teste que possuem *threads* orientadas à E/S e a longos períodos de computação, o tempo de resposta chega a ser, em média, 63% inferior ao apresentado por *kernels* baseados em eventos.

Palavras-chave: redes de sensores sem fio, sistemas operacionais, algoritmos de escalonamento.

Abstract

The ever-increasing developments of low-power integrated circuits have made it possible the design of very small low-cost and low-power electronic sensors with wireless communication and computing capabilities. Those devices, in their turn, made it feasible the implementation of the so-called Wireless Sensors Networks (WSN). WSN is a network of such devices (known as nodes), each one having an embedded microcontroller and a communication module which makes it possible the nodes to be used as sensors which process and exchange information with the other nodes, in order to achieve a specific purpose. Usually, due to the nodes very limited processing power, a very simple operating system (SO) is used to manage the node's processing and communicating capabilities by executing tasks in a concurrent fashion. The SO is a very important part in the design of a WSN and, depending on the concurrence model used on its design, the SO can be divided into two types: event-based or thread-based SO's. Event-based models make it difficult for the programmer to control the execution flow and are not suitable for tasks with long computation time. Thread-based models, on the other hand, present heavy memory use, but have a much simpler programming model and good real-time responses. In this sense, this dissertation proposes a new semi-preemptive SO, called SKMotes has the relatively easy-programming model related to thread-based models and a low memory usage. Despite SKMotes be thread-based, it is not fully preemptive, since at any given time, only a subset of the system's threads is executing as preemptive priority-based tasks and the rest of them remains on hold, which makes for low context usage, since the threads do not need data stack. This approach provides low time response for high-priority threads while at the same time guarantees lower memory usage than that of preemptive kernels. These features make SKMotes very suitable for WSN applications where there is a combination of I/O-oriented tasks and task with long computation times (for example,

applications that perform data compression and/or cryptography). After being implemented, SKMotes' performance analysis was carried out by using a specially-designed FPGA-based module, which made it possible to perform CPU-usage measurements as well as threads' time response, with the system on the fly. The measurement's results showed that, for the considered test-scenario, SKMotes presents CPU-usage rates equal to preemptive multi-threading approaches but having a lower memory usage (20

Keywords: wireless sensor networks, operating systems, schedulling algorithms.

Agradecimentos

Dedico meus sinceros agradecimentos para:

- a minha família por ser a base na qual sustento as minhas ações;

- a minha esposa Waneska, pelo seu amor, carinho, compreensão e auxílio durante a execução deste trabalho;

- o meu orientador Prof. Dr. Paulo Cortez, por sua dedicação e conselhos;

- o meu co-orientador Prof. Dr. Helano Castro, por suas valorosas contribuições e apoio durante toda essa caminhada;

- os colegas professores do IFCE campus Maracanaú, pelo incentivo.

Capítulo 1

Introdução

Os recentes avanços nas tecnologias de sistemas micro-eletro-mecânicos, circuitos integrados de baixa potência e comunicação sem fio de baixa potência permitiram a criação de pequenos dispositivos computacionais dotados de comunicação sem fio e alimentados por bateria, que podem ser organizados em rede com o intuito de realizar tarefas complexas, como: monitoramento de ambientes; rastreamento de alvos; detecção de eventos; dentre outras. Essas Redes de Sensores sem Fio (RSSFs) têm pontencial para integrar o mundo real com a infraestrutura de comunicação existente, permitindo a criação de um amplo espectro de aplicações, apenas para citar algumas: operacões de combate a tragédias naturais; mapeamento da biodiversidade; edificações inteligentes; agricultura de precisão; medicina e saúde.

1.1 Motivação

Projetar aplicações para RSSFs é um grande desafio. Um desenvolvedor de *software* para nós sensores deve encarar plataformas de *hardware* com recursos escassos e que, muitas vezes, foram especialmente projetadas para a aplicação em questão. Um sistema operacional (SO) é a camada de *software* que reside entre o *hardware* simples do sensor e as aplicações do programador. Os SOs para nós sensores são muito mais simples e oferecem muito menos recursos que os similares utilizados em outras aplicações computacionais.

Além de esconder os detalhes do *hardware*, um SO deve fornecer uma estrutura na qual o programador possa criar sua aplicação a partir de unidades executáveis, que sejam capazes de cumprir as obrigações de comunicação, amostragem e controle do nó sensor. Além disso, um SO deve implementar políticas que vão ao encontro do consumo de energia eficiente, dessa forma prolongando o tempo de vida da aplicação.

Ao longo dos anos, diversos SOs para RSSFs surgiram para facilitar o desenvolvimento de aplicações. Em relação ao modelo de concorrência, podem-se dividir os SOs em duas categorias: baseados em eventos e baseados em *threads*. O modelo baseado em eventos tende a fornecer uma solução com baixo consumo de memória, com boa gestão de energia e baixo tempo de resposta aos eventos, mas cria dificuldades ao programador para controlar os fluxos de execução e não se ajusta a problemas com longos períodos de computação, como: compressão de dados e criptografia. O modelo baseado em *threads* tem alto consumo de memória, mas fornece um modelo de programação mais simples e usual.

O TinyOS (HILL et al., 2000), em conjunto com a linguagem nesC (GAY et al., 2003), é um dos SOs mais utilizados em aplicações de RSSF. O modelo de programação empregado no TinyOS é o modelo baseado em eventos, no qual os programas são estruturados como coleções de gerenciadores de eventos que reagem aos eventos gerados pelo ambiente (MCCARTNEY; SRIDHAR, 2006). Neste caso, a ação de programar é, então, escrever rotinas curtas que respondem a esses eventos.

Outro sistema operacional baseado em eventos para nós sensores é o Contiki (DUNKELS; GRONVALL; VOIGT, 2004), que permite a reconfiguração dinâmica das aplicações e prover bibliotecas para implementação de *multithreading* cooperativo ou preemptivo sobre o sistema baseado em eventos (DUNKELS et al., 2006).

O Mantis OS (BHATTI et al., 2005) implementa um *multithreading* preemptivo tradicional em nós sensores. Para permitir o paradigma de programação no estilo *thread*, o *kernel* do Mantis oferece uma biblioteca de E/S síncrona e primitivas de controle de concorrência.

Nesse contexto, propõe-se um modelo de concorrência baseado em *threads*, mas não completamente preemptivo, pois em dado momento apenas um subconjunto das *threads* do sistema deve executar no modo preemptivo baseado em prioridades. O restante das *threads* permanece em espera, ocupando apenas um contexto mínimo de execução, que não contempla pilha de dados. Essa abordagem é alcançada através da utilização de chamadas de sistemas especiais que bloqueiam as *threads* que esperam por eventos específicos, de forma que o contexto de execução seja apenas o ponto de continuação na rotina principal da *thread*. O principal objetivo deste modelo é prover tempos de resposta baixos para *threads* de alta prioridade, ao mesmo tempo garantir baixa ocupação de memória e baixo consumo de energia. Estas características permitem que o SKMotes seja empregado em aplicações de RSSFs que empreguem simultaneamente tarefas orientadas à E/S e a longos períodos de computação. Por exemplo, aplicações de RSSFs que realizem funções de

compressão de dados (TAVLI; BAGCI; CEYLAN, 2010), criptografia (WANDER et al., 2005), dentre outras, que demandem longos períodos de computação.

1.2 Objetivos

A presente dissertação tem como objetivo geral propor um novo sistema operacional para RSSFs, baseado em *threads*, com baixos requisitos de memória e baixa latência de execução para *threads* de alta prioridade, bem como avaliar o desempenho deste novo SO. No decorrer do desenvolvimento deste trabalho, os seguintes objetivos específicos foram perseguidos:

1. avaliar a organização e os modelos de concorrência empregados nos SOs para RSSFs;
2. implementar o SO proposto em uma plataforma de *hardware* embarcado;
3. implementar um ambiente de testes baseado em dispositivo lógico programável (PLD) para auxiliar na avaliação de desempenho do SO proposto;
4. avaliar o desempenho do SO proposto em relação aos SOs utilizados em aplicações reais.

1.3 Resumo das Contribuições

Este projeto visa agregar as seguintes contribuições para o desenvolvimento de sistemas operacionais para nós sensores, mais especificamente no quesito de modelos de concorrência:

1. concepção, implementação e avaliação de um SO para RSSF;
2. concepção, implementação e avaliação de um modelo de concorrência voltado para dispositivos embarcados de rede com poucos recursos;
3. concepção e implementação de um ambiente de testes baseado em dispositivo lógico programável (PLD), capaz de realizar medições quantitativas das métricas de escalonamento de um nó de RSSF;
4. análise comparativa do desempenho da proposta em relação a soluções consagradas.

Durante a execução desta dissertação, foi publicado um trabalho completo no congresso WSE 2010 (Workshop de Sistemas Embarcados) enquanto as demais contribuições serão submetidas para um periódico (JÚNIOR; CASTRO; CORTEZ, 2010);

1.4 Organização

Uma vez introduzida a motivação deste trabalho e os objetivos a que se propõe, expõe-se a seguir como o restante está organizado. No Capítulo 2, são descritas as principais características das RSSFs.

No Capítulo 3, apresenta-se uma pesquisa das principais características dos SOs para RSSFs e dos principais SOs disponíveis para utilização.

O SKMotes é apresentado no Capítulo 4, juntamente com sua avaliação de desempenho e comparação com outros SOs.

As conclusões deste trabalho e os trabalhos futuros são apresentados no Capítulo 5.

Capítulo 2

Redes de Sensores sem Fio

O objetivo deste Capítulo é apresentar os conceitos e definições relacionados com as Redes de Sensores sem Fio (RSSFs). Na seção seguinte, apresentam-se uma visão geral das RSSFs. A seção 2.2 descreve a estrutura básica de uma RSSF. Em seguida, na seção 2.3, a arquitetura do nó sensor é apresentada. A seção 2.4 examina as características dos principais serviços providos pelos nós de uma RSSF. Na seção 2.5, são discutidos alguns dos principais protocolos de comunicação aplicados nessa área. Adiante, na seção 2.6, alguns exemplos reais de RSSFs são mostrados. Por fim, a seção 2.7 conclui o tópico estudado nesse Capítulo.

2.1 Introdução

RSSFs são fruto dos recentes avanços nas tecnologias de sistemas micro-eletro-mecânicos, circuitos integrados de baixa potência e comunicação sem fio de baixa potência. Esses avanços permitiram a criação de minúsculos dispositivos computacionais de baixo custo e baixa potência, capazes de monitorar grandezas físicas do ambiente e estabelecer comunicação uns com os outros. Esses dispositivos, denominados nós sensores, são dotados de um microcontrolador simples, elementos sensores, rádio transceptor e fonte de alimentação (CULLER; ESTRIN; SRIVASTAVA, 2004).

Uma RSSF é formada por um conjunto de nós sensores, cujo tamanho pode variar de alguns poucos nós até milhares destes, implantados em uma região de interesse com o intuito de realizar tarefas de monitoramento do ambiente. Este monitoramento obtém dados que são disponibilizados através de uma conexão com uma rede externa, tal como a Internet.

Uma RSSF é projetado para funcionar por longos períodos de tempo sem a assistência

de operadores. Contudo, a integração dos sensores num certo ambiente físico os expõem a diversos tipos de agentes agressores, como: intempéries climáticas; ação de animais e de vândalos; dentre outros. Por outro lado, uma RSSF pode sofrer frequentes mudanças na sua topologia devido à falhas nos nós, que são contornados pela redundância dos nós ou adição de novos nós na rede em operação. Assim, é desejável que as RSSFs sejam autônomas, isto é, seus nós devem ser capazes de se auto-organizarem, estabelecerem novas rotas e, assim, tolerarem falhas em um determinado número de nós da rede (ZOU; CHAKRABARTY, 2007).

Os nós sensores de um RSSF são dispositivos de rede simples em comparação aos dispositivos empregados em outros tipos de redes de comunicação sem fio. Essa simplicidade é a principal razão do baixo custo desses componentes, mas isto impõe sérias restrições no poder de processamento, de comunicação e de consumo de energia. A mais importante das restrições é a de consumo de energia, pois em boa parte das aplicações os nós são implantados em locais de difícil acesso, impossibilitando quase totalmente a substituição de suas baterias. Por isso, fontes de energia secundárias, que capturam energia do ambiente como painéis solares, podem ser acoplados aos nós, dependendo da aplicação (SEAH; EU; TAN, 2009). Além disso, a energia pode ser conservada através do emprego de inúmeras técnicas, como: roteamento de múltiplos saltos (GAO et al., 2009); agregação de dados (LIN; YEN; LIN, 2011); eliminação da redundância dos dados (VIDHYAPRIYA; VANATHI, 2009); operação com baixos ciclos ativos (GUO et al., 2009); dentre outras.

As limitações dos nós da RSSF exigem uma abordagem de projeto diferenciada em relação a outras redes de comunicação sem fio. Desta forma, pode-se entender uma RSSF como um sistema computacional dedicado, cujos requisitos da aplicação alvo tem influência imperativa na seleção e projeto dos componentes de *hardware*, algoritmos, protocolos, sistemas operacionais e *middlewares* utilizados. Um exemplo dessa característica são os protocolos de roteamento, que são centrados em dados. Nesses protocolos, a descrição e necessidade de um certo tipo de dado influencia, em conjunto com as restrições de consumo de energia e alcance do enlace de comunicação, o processo de definição de rotas. Além disso, é comum o emprego de técnicas de processamento de dados interno à rede, ou seja, ao longo do encaminhamento das mensagens, dados coletados por diferentes sensores são agregados, como uma forma de reduzir a quantidade de mensagens que trafega pela rede e consequentemente seu consumo de energia (GABER; RöHM; HERINK, 2009).

Nos últimos anos, as RSSFs têm sido motivo de atenção da comunidade científica pelo seu pontencial de integrar diferentes dados do mundo real com a infraestrutura de comunicação existente, permitindo a criação de um amplo espectro de aplicações em diversas

áreas: indústria (CHRISTIN; MOGRE; HOLLICK, 2010); meio ambiente (CHATTERJEA et al., 2008); agricultura de precisão (WILLIAMS; BERGMANN, 2007); medicina (HANDE et al., 2007); defesa (GUI; MOHAPTRA, 2004) ; mineração (CHEHRI; FORTIER; TARDIF, 2009), dentre outras.

2.2 Estrutura de uma RSSF

Uma RSSF típica possui pouca ou nenhuma infraestrutura de comunicação. Em geral, uma rede é formada por um conjunto de nós sensores, trabalhando para monitorar uma região física de interesse. A organização básica de uma RSSF é representada na Figura 2.1. Os nós sensores são instalados na região de interesse e os dados coletados são encaminhados ao nó sorvedouro. Este nó é responsável por fazer a interface com a estação de monitoração, na qual as informações de monitoração devem ser armazenadas em uma base de dados e, posteriormente, acessadas a partir de outras redes de comunicação.

Uma RSSF pode ser analisada por diferentes aspectos da sua organização, tais como: modo de instalação; modo de coleta de dados; mobilidade; relações hierárquicas; diferenças de *hardware* dos nós sensores; dentre outros.

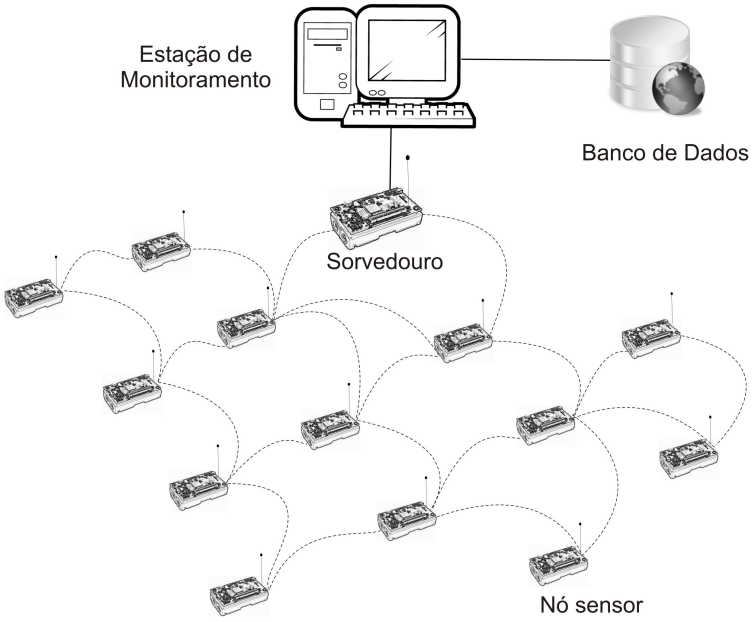

Figura 2.1: organização básica de uma RSSF.

A instalação de nós de uma RSSF pode seguir um planejamento específico, no qual determina-se o local de instalação de todos ou de alguns nós da rede. Nesta modalidade, é possível especificar o número de nós instalados pelas características da região alvo. Por outro lado, algumas aplicações não permitem um projeto de instalação prévio e os nós são

instalados no campo de forma *ad hoc*. Após a instalação, a rede é deixada sem assistência para realizar as funções de monitoramento. Normalmente, uma quantidade grande de nós é utilizada para evitar que algumas regiões fiquem sem cobertura. A Figura 2.2 mostra o processo de instalação dos nós na região de interesse, e a fase posterior de auto-organização dos nós, na qual cada nó identifica os nós presentes em sua vizinhança.

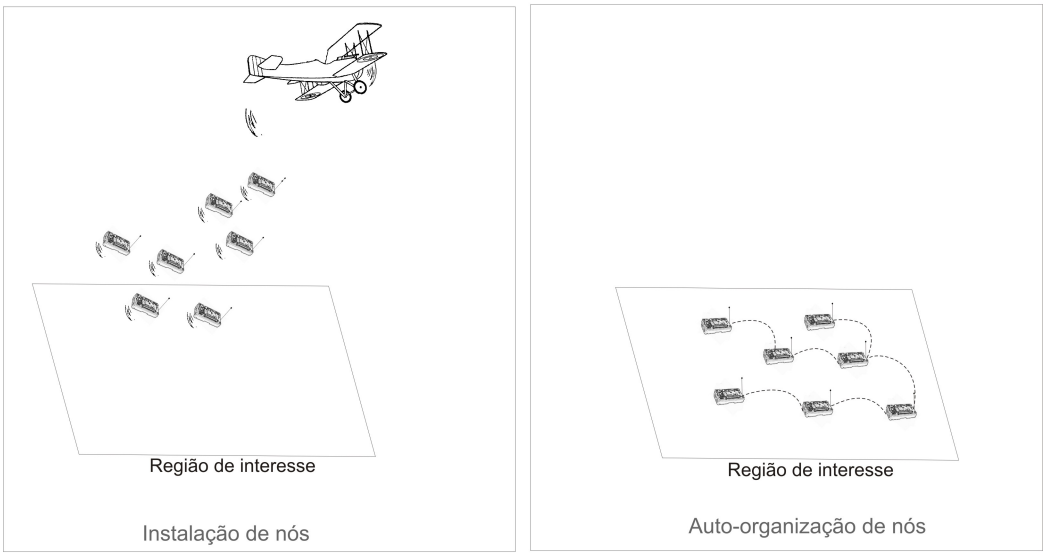

Figura 2.2: instalação e auto-organização dos nós.

Diferentes modos de coleta de dados podem ser utilizados em RSSFs, dependendo dos requisitos da aplicação. Podem-se destacar os modos: periódico; dirigido a evento e sobdemanda. No modo periódico, os nós são configurados para realizarem amostragens e envio dos resultados em um determinado período, garantindo um envio contínuo de dados. No modo dirigido a eventos, a descrição de eventos de interesse são difundidas na rede. Os nós passam a monitorar as condições específicas do evento e apenas quando tais condições são detectadas, um fluxo de comunicação é estabelecido para notificar a ocorrência do evento. Esse modo provê menor consumo de energia do que o modo periódico. Por último, o modo sob-demanda que determina que as informações sejam repassadas para a estação base após um comando específico ter sido enviado para a rede (TILAK; ABU-GHAZALEH; HEINZELMAN, 2002).

A mobilidade é um aspecto importante na organização de uma RSSF. Na maior parte das aplicações, os nós são estáticos. Contudo, algumas aplicações exigem que todos, ou alguns nós da rede, sejam móveis, como por exemplo, nós instalados em veículos ou no corpo de animais selvagens. A mobilidade dos nós requer uma análise específica da conectividade da rede e é possível que alguns nós fiquem longos períodos fora da região de cobertura. Por outro lado, a mobilidade de nós pode ser utilizada a favor da aplicação.

Por exemplo, em algumas aplicações, os nós móveis possuem modelos de movimentação determinísticos e podem ser utilizados para difundir ou agregar mensagens pela rede (TILAK; ABU-GHAZALEH; HEINZELMAN, 2002).

Uma das propriedades mais importantes das RSSFs é a auto-organização. Os nós devem ser capazes de, sem assistência externa, formarem uma estrutura de comunicação que cumpra com os requisitos da aplicação. Em alguns casos, é necessário que os nós se organizem em estruturas hierárquicas com o intuito de aproveitar a redundância das amostragens, balancear o consumo de energia e a carga computacional. Normalmente, estas estruturas são chamadas de *clusters*, os quais possuem um nó, chamado *cluster head* responsável por gerenciar as atividades do grupo (ABBASI; YOUNIS, 2007). Por outro lado, algumas redes não possuem diferenças hierárquicas entre os nós; essas redes são denominadas redes planas. A Figura 2.3 ilustra a organização desses dois tipos de redes.

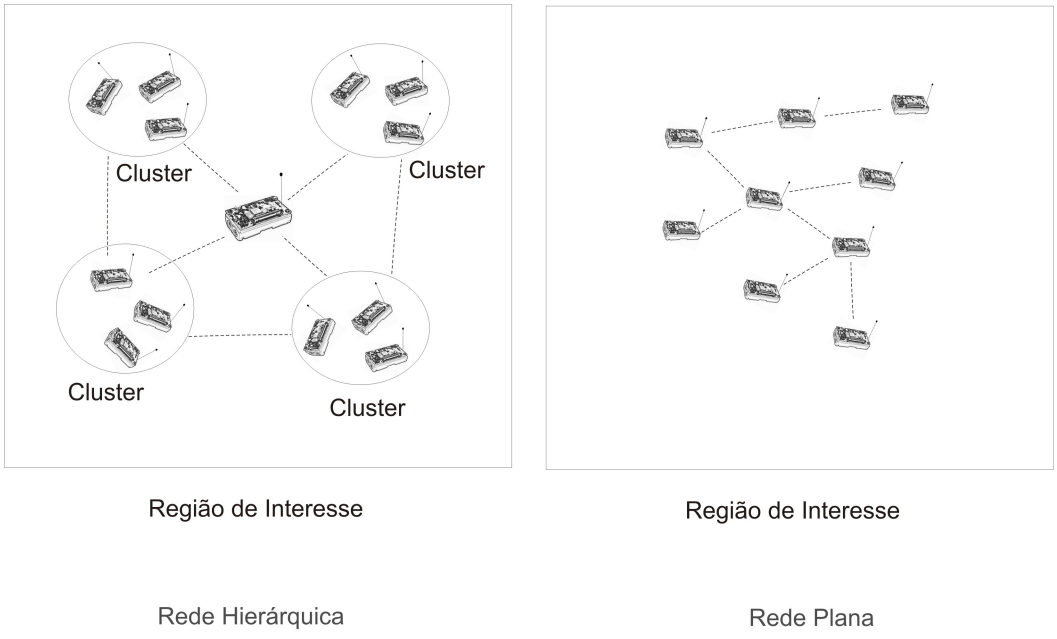

Figura 2.3: redes hierárquicas e redes planas.

As RSSFs são ditas homogêneas quando todos os nós sensores possuem a mesma configuração de *hardware*, tendo como vantagem o fato de um nó qualquer poder assumir o papel de outro nó, em caso de falhas. Entretanto, alguns destes tipos de redes se apoiam na existência de nós com diferentes capacidades. Por exemplo, uma rede hierárquica pode designar como *clusters heads* nós com *hardware* específico, normalmente com capacidades superiores de processamento ou com reservas extras de energia (ABBASI; YOUNIS, 2007).

2.3 Nó sensor

Um nó sensor é o componente fundamental de uma RSSF e todas as tarefas inerentes ao monitoramento do ambiente recaem sobre este. A Figura 2.4 ilustra a estrutura deste dispositivo. Em síntese, um nó sensor é um sistema computacional dedicado cujo projeto visa oferecer uma estrutura capaz de realizar as tarefas de amostragem dos sensores, processamento digital de sinais, comunicação e gestão dos recursos.

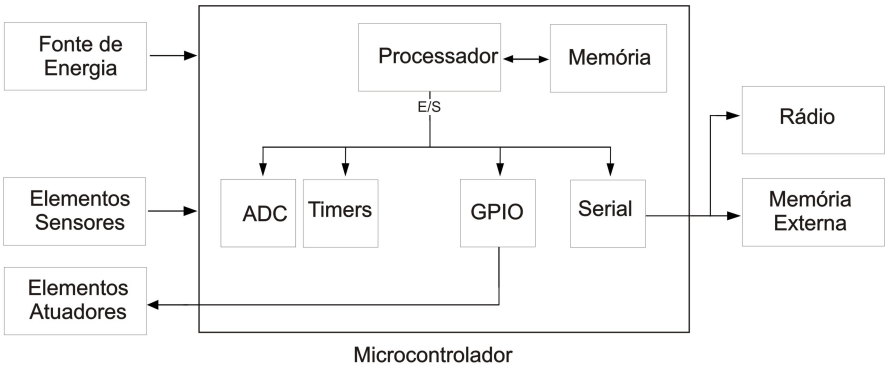

Figura 2.4: estrutura básica de um nó sensor.

Os componentes principais dessa arquitetura são: microcontrolador; memória externa; elementos sensores e atuadores; rádio transceptor e fonte de alimentação.

O microcontrolador (MCU) é um circuito integrado (CI) do tipo *System-On-Chip* (SOC), que aglutina em um único CI um pequeno sistema de computação formado por: processador de 8 bits ou 16bits; memória *flash* para armazenamento de instruções; memória RAM para armazenamento de dados durante a execução e diversos periféricos. O MCU é responsável por coordenar todas as atividades do nó sensor e por questões de custo, os MCUs utilizados possuem pouco poder de processamento e pouca memória de programa e de dados. Os MCUs possuem modos de execução de baixo consumo de energia, no qual parte dos circuitos são desligados, permitindo uma maior conservação de energia; a gestão dos modos de execução é atribuída ao *software* de controle do nó sensor.

A interface do MCU com o ambiente externo é realizada através dos dispositivos periféricos. Os periféricos mais utilizados são: conversor analógico-digital (ADC); portas de entrada e saída (GPIO); temporizadores e os canais de comunicação serial. Os ADCs são utilizados para realizar a conversão do sinal elétrico proveniente dos elementos sensores em uma representação digital passível de processamento pelo MCU. Normalmente os ADCs utilizados possuem entre 8 e 12 bits de resolução. Os GPIOs são utilizados para acionar elementos atuadores que possam existir no circuito do nó sensor, como: relés; LED; dentre outros. Os temporizadores podem ser utilizados para gerar interrupções periódicas

ou para medir o intervalo de tempo entre eventos de mudança de nível lógico ou de borda nos GPIOs. Os canais de comunicação serial permitem que o MCU interaja com outros dipositivos como as memórias externas e os rádios transceptores. Usualmente, existe uma ou mais interfaces que implementam diferentes padrões de comunicação serial, como: SPI; I2C ou serial assíncrona. Todos esses dispositivos são capazes de gerar interrupções de *hardware* para notificar a ocorrência de eventos específicos. Essas interrupções obrigam o MCU a executar um código tratador do evento ocorrido.

Com o intuito de prover armazenamento persistente de informações, alguns nós sensores possuem *chips* de memória externa, normalmente conectados através de uma interface serial. Esse dipositivo pode ser utilizado como apoio durante processo de atualização em campo do programa do nó sensor.

Os elementos sensores são componentes capazes de converter um determinado fenômeno físico na forma de sinal elétrico, que pode ser lido pelo MCU através do ADC. Os tipos de elementos sensores utilizados são dependentes da natureza da aplicação da RSSF. Os atuadores são interfaces que modificam parâmetros que controlam o estado do objeto controlado. Por exemplo, em uma aplicação de monitoramento de incêndios, o nó sensor poderia acionar um alerta sonoro por meio de uma buzina eletrônica.

O rádio transceptor é o dispositivo que permite a integração entre nós sensores. A interface entre o MCU e o rádio é feita através de um canal de comunicação serial e alguns pinos de controle, que servem para notificar a ocorrência de interrupções geradas pelo rádio ou seu estado. Essa interface de comunicação garante acesso aos registradores internos do controlador do rádio, dando amplo acesso às suas funcionalidades. Cada rádio tranceptor possui diferentes características de transmissão e recepção dos sinais RF, contudo, geralmente os rádios utilizados possuem baixa taxa de transmissão de dados e baixo consumo de energia. Em alguns casos, para conservar a energia, o MCU pode desligar temporariamente o rádio.

A execução de todas essas funcionalidades do nó sensor só é possível com a existência de um componente, que proveja a energia necessária. Isto é realizado pelas baterias. Na maior parte das aplicações, não é possível recarregar ou trocar as baterias dos sensores após a instalação, por isso, todo o projeto do *hardware* e *software* do nó sensor deve ser voltado para a utilização racional das reservas de energia do sistema.

2.4 Serviços de rede

Os principais serviços de rede foram analisados por Yick et al. (YICK; MUKHERJEE; GHOSAL, 2008), e são apresentados nesta seção. Os serviços de controle, gerenciamento e provisionamento são projetados para permitir a coordenação e gerenciamento dos nós sensores. Esses serviços melhoram a performance da rede em termos de consumo de energia, distribuição de tarefas e utilização dos recursos.

O serviço de provisionamento aloca apropriadamente recursos como energia e banda de comunicação para maximizar utilização da RSSF. Neste, existem dois aspectos: a cobertura e a localização. A cobertura visa garantir que uma RSSF possa monitorar toda a região alvo com certo grau de confiabilidade. Cobertura é importante porque afeta o número de sensores a serem instalados, o posicionamento desses sensores, conectividade e energia de uma rede.

Localização é o processo pelo qual os nós tentam descobrir sua própria localização após sua instalação. Serviços de gerenciamento e controle possuem uma grande importância por proverem suporte aos serviços de *middleware* tais como segurança, sincronização, compressão e agregação de dados, gerenciamento de energia, etc.

2.4.1 Localização

Em RSSFs, nós sensores que são instalados no ambiente de maneira *ad-hoc* não possuem conhecimento prévio de suas localizações. O problema de determinar as coordenadas da posição do nó é chamado localização. Métodos de localização correntes incluem GPS, *beacon* e localização baseada na proximidade. Equipar os nós sensores com GPS é uma solução simples para o problema. Contudo, esta solução aumenta os custos e problemas de funcionamento em alguns ambientes.

O método da localização do tipo *beacon* faz uso de nós fárois, que possuem conhecimento de sua localização para ajudar os sensores a identificarem sua posição. Entretanto, este método possui alguns problemas, sendo que o principal destes é não funcionar corretamente em redes de grande escala. Por outro lado, o método da localização baseada em proximidade faz uso dos nós vizinhos para determinar a localização, e após realizar a operação, o nó começa a se comportar como um *beacon*.

Uma outra abordagem, apresentada por Boukerche et al. (BOUKERCHE et al., 2008), é a utilização de nós *beacon* móveis, que percorrem o campo de sensoriamento compartilhando suas informações de localização e sincronismo.

2.4.2 Cobertura

Dada uma RSSF, o problema de determinar a cobertura de uma dada região é importante quando se deseja avaliar a eficiência de uma RSSF. A qualidade na monitoração depende da aplicação da rede. Aplicações como rastreamento de alvos podem requerer um alto grau de cobertura para rastrear com precisão a posição do alvo, por outro lado aplicações como monitoração ambiental pode tolerar um grau menor de cobertura. Um alto grau de cobertura requer vários sensores monitorando a mesma localização para produzir resultados confiáveis. Por isso, as pesquisas atuais focam na cobertura aplicada no contexto da conservação de energia. Algumas dessas pesquisas propõem técnicas que selecionam o conjunto mínimo de nós ativos necessário para manter uma certa cobertura.

2.4.3 Sincronização

Sincronização é importante para tarefas como roteamento e conservação de energia. A falta de precisão na medição de tempo pode reduzir significativamente o tempo de vida de uma RSSF. A sincronização de tempo global auxilia os nós a cooperarem e transmitirem dados de maneira organizada. Energia é conservada com a diminuição das colisões e retransmissões de pacotes. Ademais, a energia é poupada quando os nós trabalham com baixos ciclos ativos. Os protocolos de sincronização visam explorar estimação de incertezas nas medições e sincronizar o relógio local de cada nó da rede.

2.4.4 Compressão e agregação de dados

Compressão e agregação de dados reduzem os custos de comunicação e aumentam a confiabilidade na transferência de dados. Ambas as técnicas são necessárias para RSSFs que possuem grandes quantidades de dados para serem enviados pela rede. Dependendo da importância dos dados, um método pode ser superior ao outro. Compressão de dados envolve consiste em comprimir o tamanho dos dados antes de serem enviados. O processo inverso é realizado na estação base. Usualmente, são utilizadas técnicas de compressão de dados que não possuem perdas de informação, assim, todas as leituras individuais dos sensores são conservadas. Por outro lado, na agregação de dados, os dados são coletados de vários sensores e combinados através de operações matemáticas simples para serem transmitidos para uma estação base. Neste caso, o valor agregado é mais importante do que as leituras individuais dos sensores. Esse método é normalmente empregado em RSSFs baseadas em *clusters*.

2.4.5 Segurança

Uma RSSF está sujeita a ameaças e riscos. Um usuário não autorizado de uma RSSF pode comprometer o funcionamento de um nó sensor, alterar a integridade dos dados, espionar o conteúdo das mensagens, injetar mensagens falsas e desperdiçar os recursos desta rede. Existem várias restrições ao emprego de técnicas de segurança em RSSFs, devido principalmente a limitação de recursos. Projetar protocolos de segurança para RSSFs requer um profundo entendimento destas limitações.

2.4.6 Gerenciamento de Energia

Gerenciamento de energia é um tópico vital na operação de uma RSSF. Os nós sensores devem utilizar os conhecimentos sobre o estado dos nós vizinhos e dos requisitos da aplicação para gerenciar os diferentes modos de consumo de energia providos pelo seu *hardware*, com o intuito de prolongar o tempo de vida do mesmo. Durante os modos de baixo consumo de energia, um nó sensor é impossibilitado de realizar algumas de suas atividades básicas, como comunicação e/ou monitoramento de eventos. Por isso, é importante que a gestão de energia não cause impacto negativo na cobertura ou na capacidade de detecção de eventos de uma RSSF.

2.5 Protocolos

O crescente interesse na área de RSSFs propiciou a concepção de diversos protocolos específicos para essa área. Esses protocolos visam atender os requisitos das aplicações de sensoriamento, ao mesmo tempo que devem respeitar as restrições inerentes aos nós sensores. Há propostas de procotolos que tratam de diferentes aspectos de RSSFs, nesta seção são apresentados alguns protocolos divididos nas seguintes categorias: centrados em dados; baseados em *clusters* e bioinspirados.

2.5.1 Centrados em dados

É esperado que RSSFs possuam um grande número de nós, que pode ser alterado após a instalação devido a falhas ou inserção de novos nós. Esta característica torna complexa a utilização de um mecanismo de endereçamento global de nós nas RSSFs. O modelo centrado em dados difere das redes tradicionais baseadas em endereços, no qual tabelas de rotas são mantidas na camada de rede da pilha de comunicação. Além disso, a maior

parte das aplicações está interessada em obter informações confiáveis de monitoramento de uma certa região e não na identificação do sensor específico que gerou esse dado.

No modelo centrado em dados, o nó sorvedouro envia para certas regiões, requisições com a descrição dos dados desejados. Após a disseminação das requisições, os nós que possuem dados desejados os enviam ao sorvedouro. O primeiro protocolo proposto seguindo este modelo foi o SPIN (*Sensor Protocols for Information via Negotiation*) (HEINZELMAN; KULIK; BALAKRISHNAN, 1999), o qual introduziu uma fase de negociação entre os nós com o intuito de conservar energia.

Um dos protocolos muito utilizados que segue esse modelo é a difusão direcionada (INTANAGONWIWAT; GOVINDAN; ESTRIN, 2000), que se tornou um modelo para outros trabalhos.

O SPIN foi proposto para superar algumas deficiências presentes nos protocolos clássicos de *flooding* (SINGH; SINGH; SINGH, 2010). No SPIN, as informações dos nós sensores são descritas em estruturas de alto nível chamadas meta-dados. Quando um nó possui informações que deseja compartilhar com a rede, uma mensagem de anúncio é enviada aos seus vizinhos, contendo um meta-dado. Ao receber um anúncio, o nó vizinho verifica se já possui a informação anunciada. Se não, o anúncio é respondido com uma requisição de dados, que deve então ser respondida pelo nó emissor.

A Figura 2.5 mostra um exemplo do protocolo. O nó A anuncia seus dados ao nó B em (a). Em seguida, o nó B responde com uma requisição ao nó A em (b). Depois de receber os dados requisitados em (c), o nó B envia anúncios aos seus vizinhos em (d), que por sua vez enviam de volta requisições ao nó B em (e-f).

A difusão direcionada é um importante marco na pesquisa de protocolos de roteamento centrados em dados para RSSFs (INTANAGONWIWAT; GOVINDAN; ESTRIN, 2000). Neste procotolo, os dados são nomeados como pares de atributo-valor. Uma tarefa de sensoriamento é disseminada em uma rede como um interesse para um dado nomeado. A disseminação configura gradientes na rede que capturam os eventos, dados que correspondem aos interesses. Os eventos fluem em direção aos nós originadores dos interesses através de várias rotas. O sorvedouro aumenta a taxa de transmissão de uma das rotas, ou um pequeno número destas. A Figura 2.6 sintetiza o funcionamento deste protocolo.

A difusão direcionada difere do protocolo SPIN em relação a forma como as requisições são realizadas. Na difusão direcionada, o sorvedouro requisita aos nós sensores a disponibilidade de um certo tipo de dado. No protocolo SPIN, cada nó anuncia a disponibilidade de seus dados aos seus vizinhos. A difusão direcionada possui diversas vantagens,

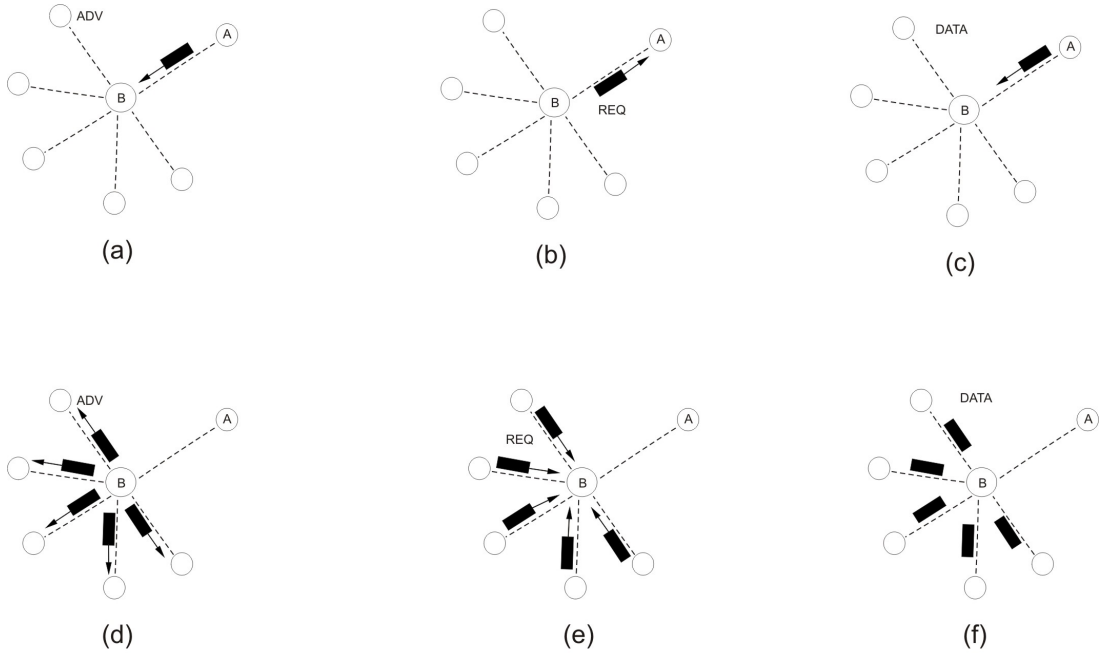

Figura 2.5: funcionamento do SPIN, adaptado de (HEINZELMAN; KULIK; BALAKRISHNAN, 1999).

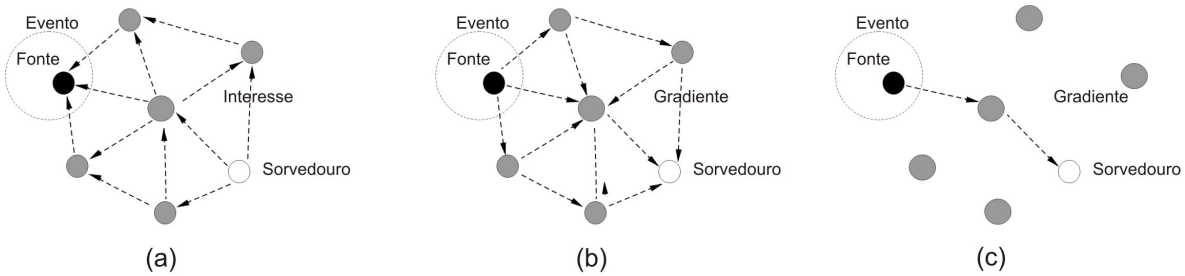

Figura 2.6: Fases do protocolo Difusão Direcionada. (a) Propagação dos interesses, (b) criação dos gradientes, (c) reforço de uma rota, adaptado de (INTANAGONWIWAT; GOVINDAN; ESTRIN, 2000).

destacando-se: a comunicação que é feita apenas entre nós vizinhos, sem a necessidade de um mecanismo de endereçamento global ou de manter uma topologia específica de rede; cada nó pode realizar agregação de dados de maneira oportunística, durante a retransmissão dos eventos. Contudo, esse protocolo não pode ser aplicado quando há a necessidade de monitoramento contínuo.

2.5.2 Baseados em *clusters*

Agrupar nós sensores em *clusters* tem sido amplamente proposto pela comunidade cientifíca como maneira de permitir a escalabilidade de RSSFs (ABBASI; YOUNIS, 2007). Cada *cluster* deve possuir um nó responsável por gerenciar as atividades do grupo, este nó líder é referenciado normalmente como *cluster head* (CH). O CH pode ser eleito pelos

sensores ou pré-designado pelo projetista da RSSF. O CH pode ser apenas um dos sensores ou um nó com mais poder de computação e energia. Os CHs podem formar uma segunda camada de rede ou apenas encaminhar as mensagens para estação base.

Além do surpote a escalabilidade da rede, a utilização de *clusters* traz várias vantagens, sendo uma delas a possibilidade de concentrar várias funções no CH, como: roteamento; agregação de dados; armazenamento; dentre outras. Desta forma, reduz-se a quantidade de atividades executadas nos nós, permitindo um melhor aproveitamento dos recursos de energia. O CH pode implementar estratégias otimizadas de gerenciamento para melhorar a performance da rede e prolongar o tempo de vida dos sensores. Além disso, o CH pode escalonar as atividades dos sensores do *cluster* de forma que os sensores possam passar a maior parte do seu tempo no modo de baixo consumo, bem como as atividades de comunicação também podem ser escalonadas, reduzindo a probabilidade de colisões (ABBASI; YOUNIS, 2007).

A formação do *cluster* é tipicamente baseada no nível de energia dos sensores e de sua proximidade com o CH. O protocolo LEACH (HEINZELMAN; CHANDRAKASAN; BALAKRISHNAN, 2000) foi um dos primeiros protocolos baseados em *clusters* para RSSFs, sendo a base para vários outros protocolos, como o PEGASIS (LINDSEY; RAGHAVENDRA, 2002) e Hierarchical-PEGASIS (LINDSEY; RAGHAVENDRA; SIVALINGAM, 2001).

A ideia principal do protocolo LEACH é formar os *clusters* baseados na intensidade do sinal recebido e utilizar CH locais como roteadores para o nó sorvedouro. Isto irá economizar energia, visto que as transmissões serão realizadas apenas pelo CH. Além disso, todo o processamento dos dados do *cluster* é realizado pelo CH. Para balancear o consumo de energia, os CHs mudam com o tempo de forma aleatória. Contudo, esse protocolo presume que todos os nós possuam comunicação direta com o nó sorvedouro.

O protocolo PEGASIS é superior ao protocolo LEACH, em vários aspectos, tais como: ao invés de criar vários CHs, este protocolo permite criar cadeias de nós sensores, de forma que cada nó da cadeia se comunica com um nó vizinho; nesse caso, um nó incorporará o papel de líder e enviá os dados ao nó sorvedouro; os dados coletados são transmitidos de nó em nó, agregados e finalmente alcançam o nó sorvedouro. A principal diferença entre este protocolo, em relação ao LEACH, é o uso de comunicação multi-salto através da formação de cadeias e a escolha de um único nó para transmissão para a estação base. Este protocolo tem performance superior ao LEACH em torno de 100%-300% para diferentes configurações de redes. Contudo, o protocolo apresenta desvantagens: introduz muitos atrasos nas transmissões para nós distantes na cadeia e usa um CH único que pode se tornar um gargalo.

2.5.3 Bioinspirados

Vários protocolos baseados em sistemas biológicos podem ser aplicados em RSSFs. Esses protocolos utilizam regras relativamente simples e informações locais para construir um consenso global de algum parâmetro da rede, como o tempo no algoritmo *firefly* (BREZA; MCCAN, 2008), o cálculo do valor médio das amostras no algoritmo *Push-Sum* (KEMPE; DOBRA; GEHRKE, 2003), ou a rota ótima entre dois grupos de nós como o protocolo EEABR (CAMILO et al., 2006), baseado na metaheurística da colônia de formigas.

O algoritmo de sincronização *firefly* é baseado na utilização de uma janela de escuta. Quando um nó inicia, este espera por uma mensagem de *broadcast* de seus vizinhos. Após receber as mensagens, um nó modifica sua referência de relógio para a média das referências de relógios anunciadas nos pacotes de seus vizinhos, e, no final,a janela de escuta de todos os nós deve sincronizar.

Push-Shum é um protocolo que implementa uma técnica de agregação de dados baseada na comunicação *gossip*. Assim, cada nó mantém duas quantidades: um peso e um agregado. Este protocolo trabalha em iterações repetidas, em cada iteração, cada nó escolhe aleatoriamente um nó da rede e envia metade de seu peso e de seu agregado e o nó receptor adiciona essas quantidades às suas próprias. Se cada nó da rede for capaz de, aleatoriamente, contactar qualquer outro nó, o algoritmo calcula o valor agregado das medições da rede em um tempo que é proporcional ao logaritmo do número de nós. O cálculo do valor agregado é realizado de forma completamente descentralizado, mas para redes maiores o tempo de convergência do algoritmo pode se tornar um empecilho.

EEABR (*Energy-Efficient Ant-based Routing Algorithm*) usa uma colônia de formigas artificiais que navegam pela RSSF, procurando caminhos entre os nós sensores e um nó destino, que são otimizados na distância e no consumo de energia, ajudando a aumentar o tempo de vida da rede. Cada formiga escolhe qual é o próximo nó que deve visitar através de uma função baseada no nível de energia do nó e na quantidade de feromônio depositado neste. Quando atinge o nó destino, a formiga viaja de volta pelo caminho construído e atualiza o nível de feromônio em uma quantidade baseada no nível de energia e no número de nós dessa rota.

2.6 Aplicações

RSSFs são uma tecnologia com o potencial de modificar drasticamente a vida cotidiana das pessoas, pois o nível de integração possível entre os sistemas de computação distribuídos e os dados do ambiente é pujante, permitindo toda uma variedade de aplicações que seriam improváveis há poucos anos atrás. Áreas que podem se beneficiar dessa tecnologia são inúmeras, apenas para citar algumas: operações de combate a tragédias naturais; mapeamento da biodiversidade; edificações inteligentes; agricultura de precisão; medicina e saúde.

Um dos mais célebres projetos de aplicação de RSSFs na monitoração de animais no habitat natural é a rede implementada na *Great Duck Island* (MAINWARING et al., 2002), a qual é localizada a 15 km ao sul da *Mount Desert Island*, Maine nos Estados Unidos. O objetivo do projeto é monitorar o comportamento de uma espécie de ave marinha que habita a ilha. O projeto possui uma série de características interessantes, como: arquitetura de rede hierárquica; possibilidade de longos períodos de operação sem manutenção das baterias; gerenciamento remoto; dentre outras. O emprego de RSSFs no monitoramento de animais traz uma grande vantagem, pois minimiza a presença dos pesquisadores no ambiente, o que permite visualizar o real comportamento dos espécimes.

Outra aplicação bem sucedida de RSSFs no monitoramento ambiental é o projeto ZebraNet (JUANG et al., 2002), que permitiu a observação do comportamento de animais selvagens no Centro de Pesquisas Mpala no Quênia. O interesse do projeto é o comportamento individual dos animais, interações entre espécies e o impacto do desenvolvimento humano nessas espécies. Os animais são equipados com os nós sensores. Um receptor GPS é integrado em cada nó para fornecer a posição e velocidade estimadas. Cada nó armazena as medições de seus sensores em intervalos de 3 minutos. Em intervalos regulares, uma estação base móvel se movimenta pela região, coletando as informações armazenadas nos nós sensores.

RSSFs também podem ajudar a entender melhor o comportamento de vulcões ativos. Pesquisadores de universidades americanas e equatorianas instalaram uma RSSF nos vulcões Tungurahua e Reventador no Equador (WERNER-ALLEN et al., 2006). RSSFs podem dar uma grande contribuição à comunidade de pesquisas geofísicas. O estudo de vulcões ativos exige alta taxas de dados, alta fidelidade das amostras, grande espaço físico entre os nós. A interseção desses requisitos com as capacidades atuais das RSSFs criam novos desafios que exigem esforço de pesquisa e desenvolvimento.

Em um vinhedo do Oregon, Estados Unidos, uma RSSF é empregada para monitorar

as condições que influenciam o crescimento das plantas (BECKWITH; TEIBEL; BOWEN, 2004). Os objetivos do projeto incluíam suporte à colheita precisa, proteção contra geadas, e desenvolvimento de novos modelos de agricultura. Na primeira versão do sistema, os nós foram instalados pelo vinhedo em uma grade regular com 20 metros de distância entre cada sensor. Os nós sensores formam uma rede multi-salto de duas camadas.

2.7 Conclusão

Nos últimos anos, a comunidade científica vem dedicando investigações na tecnologia de RSSFs. Diversos protocolos, *middlewares*, sistemas operacionais, ferramentas de um modo geral, foram propostos para ajudar a solucionar os desafios pertencentes a essa tecnologia e abrir espaço para todo tipo de aplicações que dela possam se beneficiar. Contudo, várias questões de pesquisa ainda estão em aberto. Devido a sua natureza, as RSSFs são bastante ligadas aos requisitos da aplicação alvo, exigindo um alto grau de adaptação das soluções de *hardware* e *software* ao contexto da aplicação.

A diversidade de ferramentas se torna um desafio aos projetistas, que devem ter ciência das propostas prévias para poder especificar e projetar suas aplicações. Por isso, é importante termos suporte de *software* básico para diferentes configurações de *hardware* de nós sensores. De forma a gerenciar os recursos de *hardware* e as aplicações que são executadas nos nós, faz-se necessário o uso de um sistema operacional que contemple os requisitos de tempo real inerente a essas aplicações.

Capítulo 3

Sistemas Operacionais para Redes de Sensores sem Fio

Ne ste Capítulo são descritas as principais características de um sistema operacional (SO) para RSSFs. Na seção 3.1, apresenta-se uma visão geral dos SOs para RSSFs. Logo em seguida, a seção 3.2 examina os desafios enfrentados pelo projetista desses sistemas. A seção 3.3 apresenta os principais componentes de um SO. A seção 3.4 detalha os modelos de concorrência empregados em aplicações de RSSFs. Em seguida, as principais propostas de SOs para esse nicho de aplicação são descritas e comparadas. Por fim, na última seção, é apresentada a conclusão deste Capítulo.

3.1 Introdução

As limitações dos nós de uma RSSF exigem uma abordagem de projeto diferenciada em relação a outras redes de comunicação sem fio. Pode-se entender uma RSSF como um sistema computacional dedicado, cujos requisitos da aplicação alvo têm influência imperativa na seleção e projeto dos componentes de *hardware*, algoritmos, protocolos, sistemas operacionais e *middlewares* utilizados.

Um projetista de SOs para nós sensores enfrenta o desafio de criar uma estrutura de *software* para plataformas computacionais simples, mas que forneça um modelo de concorrência adequado com baixo consumo de energia e memória. Este modelo de concorrência define como os programadores devem organizar seu código em unidades executáveis pelo SO e como estas unidades devem compartilhar os recursos do sistema, inclusive o processador.

Nós sensores típicos são equipados com microcontroladores de 8-bit ou 16-bit, com

memória de programa de dezenas de kilobytes e alguns kilobytes de memória para dados. Além disso, uma série de periféricos são utilizados para auxiliar nas funções de nó, como: temporizadores; conversores analógico-digital; portas de entrada e saída; interfaces de comunicação serial; dentre outros. O SO deve gerenciar esses periféricos e fornecer uma interface simples para sua utilização.

As funções básicas de um SO incluem abstração dos recursos de *hardware*, gerenciamento das interrupções e escalonamento das tarefas, controle da concorrência e suporte a comunicação em rede. A implementação dos serviços listados não é trivial por causa das severas restrições do *hardware* do nó sensor. Ao contrário dos SOs para outros sistemas de computação, o SO de RSSFs é essencialmente uma biblioteca, ligada ao código da aplicação para produzir um binário para execução (MOTTOLA; PICCO, 2011).

Ao longo dos anos, diversos SOs para RSSFs surgiram para facilitar o desenvolvimento de aplicações. Em relação ao modelo de concorrência, podem-se dividir os SOs em duas categorias: baseados em eventos e baseados em *threads*. O modelo baseado em eventos tende a fornecer uma solução com baixo consumo de memória, com boa gestão de energia e baixos tempos de resposta aos eventos, mas cria dificuldades ao programador para controlar os fluxos de execução e não se ajusta a tarefas com longos períodos de computação. Já o modelo baseado em *threads* tem alto consumo de memória, mas fornece um modelo de programação mais simples e usual.

O TinyOS (HILL et al., 2000), em conjunto com a linguagem nesC (GAY et al., 2003), é um dos SOs mais utilizados em aplicações de RSSF. O modelo de programação empregado no TinyOS é um modelo baseado em eventos, no qual os programas são estruturados como coleções de gerenciadores de eventos que reagem aos eventos gerados pelo ambiente. A ação de programar é, então, escrever rotinas curtas que respondem a esses eventos (MCCARTNEY; SRIDHAR, 2006).

Outro sistema operacional baseado em eventos para nós sensores é o Contiki (DUNKELS; GRONVALL; VOIGT, 2004), que permite a reconfiguração dinâmica das aplicações e provê bibliotecas para implementação de *multithreading* cooperativo, ou preemptivo sobre o sistema baseado em eventos (DUNKELS et al., 2006).

Um SO que se destaca na área de RSSFs é o Mantis OS (BHATTI et al., 2005). Este SO implementa um *multithreading* preemptivo tradicional em nós sensores. Para permitir o paradigma de programação no estilo *thread*, o *kernel* do Mantis oferece uma biblioteca de E/S síncrona e primitivas de controle de concorrência.

Além desses sistemas, podem-se destacar algumas propostas que estendem o modelo

de concorrência de SOs já citados, adicionando características de outros modelos de concorrência. Por exemplo, TOSThreads (KLUES et al., 2009); TinyThreads (MCCARTNEY; SRIDHAR, 2006) e TinyMOS(TRUMPLER; HAN, 2006) adicionam novas funcionalidades ao TinyOS. Assim como, a biblioteca Protothreads permite adicionar o estilo de progamação *thread* no Contiki (DUNKELS et al., 2006).

3.2 Desafios em RSSFs

Um projetista de SOs para aplicações de RSSFs deve enfrentar vários desafios, a maioria deles relacionados com as severas restrições de recursos do *hardware* do nó sensor e os requisitos das aplicações. Os maiores desafios que influenciam o projeto de um SO são: baixa disponibilidade de memória; garantia de tempo real; consumo eficiente de energia; confiabilidade; reconfigurabilidade e conveniência de programação (DONG et al., 2010).

Baixa Disponibilidade de Memória

Nós sensores possuem apenas alguns kilobytes de memória disponível, exigindo que o SO seja projetado para ocupar pouco espaço. Essa é uma característica fundamental dos SOs para redes de sensores e também justifica a exclusão de SO embarcados mais sofisticados nesse nicho de aplicação.

Garantia de Tempo Real

Algumas aplicações exigem amostragem de alta fidelidade. Por exemplo, monitoração de terremotos requer amostragem precisa com frequência de 100 Hz (SARUWATARI; SUZUKI; MORIKAWA, 2009). Garantia de tempo real é um requisito necessário nessas aplicações. Assim, um SO deve garantir que tarefas com requisitos de tempo tenham prioridade em relação a outras tarefas do sistema.

Consumo Eficiente de Energia

Nós sensores possuem alimentação por baterias que define uma vida útil limitada. Por outro lado, várias aplicações requerem que os nós operem por longos períodos sem assistência externa. Esse requisito implica que um SO deve prover mecanismos de conservação da energia com o intuito de prolongar o tempo de vida dos sensores. Isto pode ser implementado pelo SO através da utilização dos modos ociosos do processador. O

tempo ocioso disponível pode ser utilizado para colocar o processador no modo de baixo consumo.

Confiabilidade

Na maior parte das aplicações, os nós são instalados uma única vez e têm a missão de operar por longos períodos sem assistência. A confiabilidade de SOs para RSSFs é de grande importância para facilitar o desenvolvimento de softwares complexos, assegurando o correto funcionamento das aplicações.

Reconfigurabilidade

Diversas aplicações contam com um número relativamente grande de nós sensores instalados em localizações de difícil acesso. Entretanto, é desejável que o sistema permita atualização dinâmica do sistema através da rede. Assim, os SOs devem fornecer mecanismos que facilitem a reprogramação dos nós.

Conveniência de Programação

Aplicações para RSSFs são diversas e apresentam vários requisitos. Logo, conveniência de programação é de grande importância para encurtar o ciclo de desenvolvimento de aplicações. Um SO deve prover estruturas que ajudem o programador a lidar com a concorrência inerente das aplicações e as complexidades das plataformas de *hardware*.

3.3 Componentes

Os componentes mais importantes que fazem parte de um SO para RSSFs são: escalonador de tarefas; abstração de hardware; pilha de comunicação; interface com sensores e ligação dinâmica.

Escalonador de Tarefas

Um escalonador é a parte do *kernel* responsável por definir a sequência de execução das tarefas de um sistema. Este trabalho é diretamente afetado pelo modelo de concorrência empregado. Atualmente, em aplicações de RSSFs, dois modelos predominam: baseado em eventos e o baseado em *threads*. Ambos os modelos possuem vantagens e desvantagens que são estudadas na próxima seção.

Abstração do Hardware

Um componente de abstração de hardware deve fornecer uma camada de *software* que simplifique o acesso aos recursos de *hardware* e permita a mobilidade de código das aplicações para diferentes plataformas físicas.

Pilha de Comunicação

A pilha de comunicação facilita o desenvolvimento de aplicações distribuídas através da disponibilização da gestão do *chip* do rádio e de serviços de comunicação de alto nível.

Interface com Sensores

A interface com os elementos sensores deve prover aos programadores uma forma simples de acesso às leituras dos dados dos sensores. Detalhes de baixo nível relacionados com a configuração dos dispostivos devem ser transparentes ao programador.

Ligação Dinâmica

Este componente permite a ligação em tempo de execução de novos módulos, facilitando o aumento de flexibilidade na reprogramação dos sensores. Quando um módulo de software é carregado dinamicamente, o componente de ligação dinâmica aloca espaço de memória para a aplicação, liga os símbolos aos endereços físicos e copia o módulo para o espaço de memória reservado. Este mecanismo é útil para tarefa de atualização do sistema, pois, exige a transmissão apenas do módulo que deve ser ligado e não de toda a imagem da memória.

3.4 Modelos de Concorrência

Um modelo de concorrência define como os programadores devem organizar seu código em unidades executáveis por dado SO e como estas unidades compartilham os recursos do sistema, inclusive o processador. É esperado que o modelo de concorrência utilizado em nós sensores proveja um paradigma de programação que facilite o desenvolvimento de aplicações, ao mesmo tempo que respeite as limitações de recursos inerentes às RSSFs.

Em síntese, existem dois modelos de concorrência que são comumente empregados em nós sensores: baseado em evento e baseado em *threads*. Em sistemas baseados em eventos, os programadores devem manter o estado da aplicação manualmente (na forma

de variáveis globais) e lidar com algumas limitações para realizar operações de entrada e saída (E/S), que são decorrentes do fato das tarefas serem obrigadas a serem executadas até o seu término sem sofrer preempção. Desta forma, as operações de E/S devem seguir o modelo de operações split phase. Em tais operações, a requisição de um serviço de E/S e sua finalização são funções separadas. Assim, o resultado da operação de E/S é notificado através de um evento (GAY et al., 2003).

Por outro lado, sistemas baseados em *threads* permitem que os programadores utilizem o modelo de programação tradicional, utilizando *threads* (DONG et al., 2010), que é mais amigável do que o modelo baseado em eventos.

Alguns projetos tentam melhorar o modelo de programação dos sistemas baseados em eventos, adicionando bibliotecas de *threads*. Por exemplo, TinyThreads (MCCARTNEY; SRIDHAR, 2006) e TOSThreads (KLUES et al., 2009) adicionam a programação *threads* no TinyOS; Contiki suporta *multithreading* através de uma biblioteca que executa sobre o *kernel* baseado em eventos (DUNKELS; GRONVALL; VOIGT, 2004). Por outro lado, sistemas *multithreading* também podem suportar modelos baseados em eventos. Por exemplo, o TinyMOS suporta eventos do TinyOS no *kernel* do Mantis (TRUMPLER; HAN, 2006).

3.4.1 Modelos Baseados em Eventos

A programação no modelo de concorrência baseado em eventos consiste de ações, também chamadas de tarefas, que são acionadas em resposta a ocorrência de eventos. Normalmente um evento pode chamar apenas uma ação, mas uma ação pode ser chamada por diferentes eventos (KASTEN; RöMER, 2005). Uma estrutura do *kernel* chamada *dispatcher* é responsável por gerenciar a invocação de ações. É comum a utilização de filas de eventos para armazenar temporariamente os eventos que ainda não foram processados.

As ações são executadas até finalizar sua atividade, não podendo ser bloqueadas ou preemptadas por outras ações. Por isso, ações devem executar em tempo determinado, permitindo que outras partes do sistema executem. Eventos que ocorrem durante a execução de uma ação devem ser mantidos na fila para posterior processamento. Quando a fila de eventos está vazia e não há nenhuma ação a ser executada, o sistema é posto em modo de baixo consumo de energia. Para implementar uma aplicação, os programadores definem ações e as designam para determinados eventos.

Um modelo baseado em eventos possui duas vantagens (SARUWATARI; SUZUKI; MORIKAWA, 2009). Primeira vantagem, o programador não precisa se preocupar com gerenciamento

de conflitos porque todas as ações são executadas até o seu término sem serem interrompidas por outras ações, ou seja não há preempção. Esta característica também reduz o desperdício da CPU com a troca de contexto entre diferentes ações. Segunda vantagem, estes modelos podem ser implementados com relativamente poucos recursos por causa de sua estrutura simples, que consiste de uma pilha de dados, *dispatcher* e ações. Esta simplicidade aumenta a portabilidade do sistema.

Por outro lado, os modelos baseados em eventos possuem algumas desvantagens evidentes. Primeira, estes modelos de eventos são de difícil gestão conforme a aplicação cresce em complexidade (KASTEN; RöMER, 2005). Além disso, a programação baseada em eventos exige um estilo de programação fundamentada em máquinas de estado que dificulta a escrita, manutenção e depuração de programas (DUNKELS et al., 2006). Klues (KLUES et al., 2009) afirma que um modelo de concorrência baseado em eventos se ajusta a pequenas tarefas que tratam eventos de entrada e saída. Entretanto, tarefas longas não se adequam a esse paradigma, e precisam ser divididas em tarefas menores. Contudo, algumas tarefas são difíceis de serem divididas, como: compressão de dados e criptografia.

A estrutura básica de um sistema baseado no modelos de eventos é mostrada na Figura 3.1 . Neste sistema, existe apenas um contexto de execução que pode executar um evento ou uma tarefa, que são selecionados pelo *dispatcher*, que prioriza a execução dos eventos.

A maior parte dos eventos é gerada em resposta às interrupções dos dispositivos de *hardware*. Os eventos dão prosseguimento ao tratamento de dados pelas rotinas de interrupção e podem postar tarefas para posterior execução. Como as tarefas executam até sua finalização, não existe o acesso concorrente a recursos entre as mesmas. Contudo, podem ocorrer condições de disputa entre as tarefas e as rotinas que tratam os eventos. Por isso, os SOs devem fornecer primitivas que garantam a atomicidade durante o acesso a regiões críticas.

3.4.2 Modelos Baseados em *Threads*

Um modelo de programação baseado em *threads* fornece ao programador um paradigma de programação mais conveniente do que um modelo baseado em eventos. Normalmente, sistemas *multithreading* são divididos em duas categorias: preemptivos e cooperativos. Em sistemas preemptivos, cada *thread* possui um contexto de execução que consiste dos registradores do processador e pilha de dados. Uma *thread* em execução pode ser preemptada pelo escalonador, de acordo com as regras do algoritmo de escalonamento. Já em um modelo cooperativo, as *threads* liberam o processador quando não possuem mais

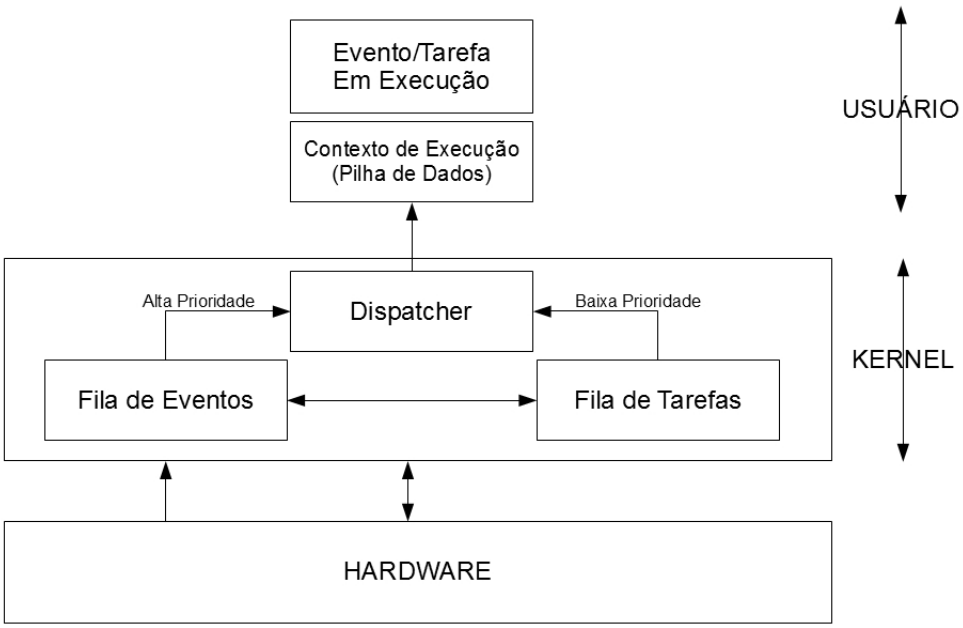

Figura 3.1: estrutura de um sistema baseado em eventos.

trabalho a ser executado.

Em ambientes com severas restrições de memória, o paradigma *multithreading* consome boa parte dos recursos de memória (DUNKELS; GRONVALL; VOIGT, 2004). Cada *thread* deve possuir sua pilha e por causa da dificuldade de dimensionar a quantidade de memória necessária para esta pilha, geralmente é sobredimensionada. Além disso, um modelo de concorrência requer mecanismos que previnam o acesso concorrente a recursos compartilhados.

É possível implementar um sistema *multithreading* cooperativo com uma única pilha de dados compartilhada por todas as *threads*. Por exemplo, a biblioteca Protothreads (DUNKELS et al., 2006) para o sistema Contiki permite a criação de um modelo cooperativo utilizando o conceito de continuações locais que captura o estado de execução dentro da função principal da *thread*, mas não armazena o estado da pilha de dados.

Contudo, essa abordagem possui algumas desvantagens, tais como não permitir utilizar variáveis locais na função da *thread*, pois estas não são salvas nas continuações locais, obrigando o progamador a usar variáveis estáticas ou globais. Além disso, o mecanismo é implementado com instruções do pré-processador C e não permite a utilização de sentenças

switch no código das protothreads.

A estrutura básica de um sistema cujo modelo é baseado em *threads* é apresentada na Figura 3.2. Cada *thread* possui seu contexto de execução próprio com pilha de dados, e o escalonador escolhe qual *thread* deve executar a partir de uma política de escalonamento. Neste tipo de modelo as chamadas de sistema são síncronas, ou seja, a *thread* é bloqueada até que o *kernel* atenda a sua requisição. Além disso, é necessário utilizar primitivas de comunicação e sincronização para criar aplicações que dependem da interação entre diferentes *threads*, para evitar os problemas clássicos de condição de disputa.

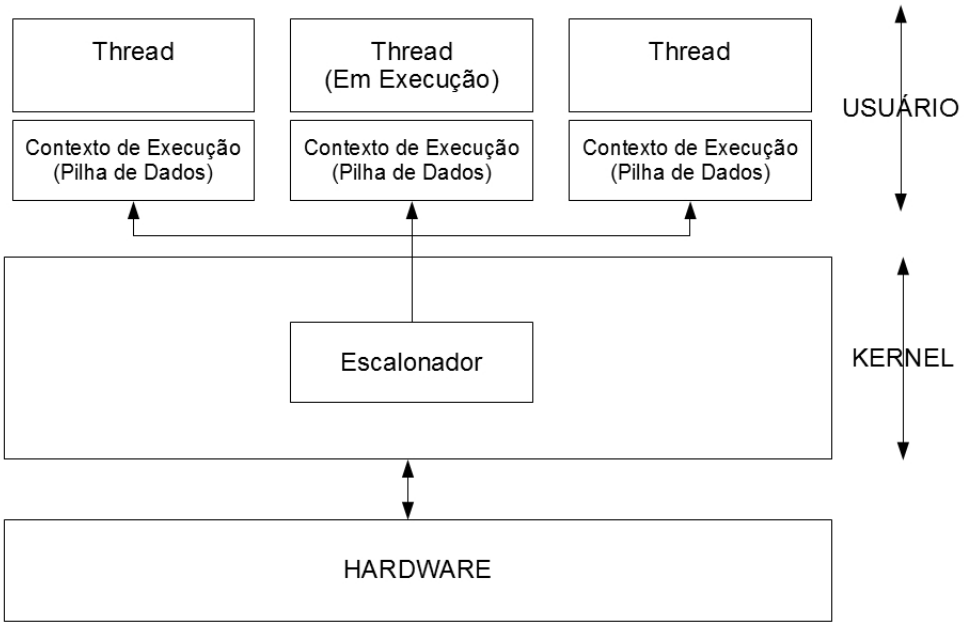

Figura 3.2: estrutura de um sistema baseado em *threads*.

Alguns sistemas mencionados oferecem a possibilidade de utilizar características dos dois modelos de concorrência. Por exemplo, é possível adicionar um *kernel* baseado em eventos em um *kernel multithreading* preemptivo. Um *kernel* baseado em eventos executaria como uma *thread* de alta prioridade, sendo responsável por tratar a maior parte dos eventos de *hardware*. A grande vantagem deste modelo é permitir que todo código já implementado para um sistema baseado em eventos, como o TinyOS, seja aproveitado em um projeto baseado em *threads*. A Figura 3.3 ilustra tal organização.

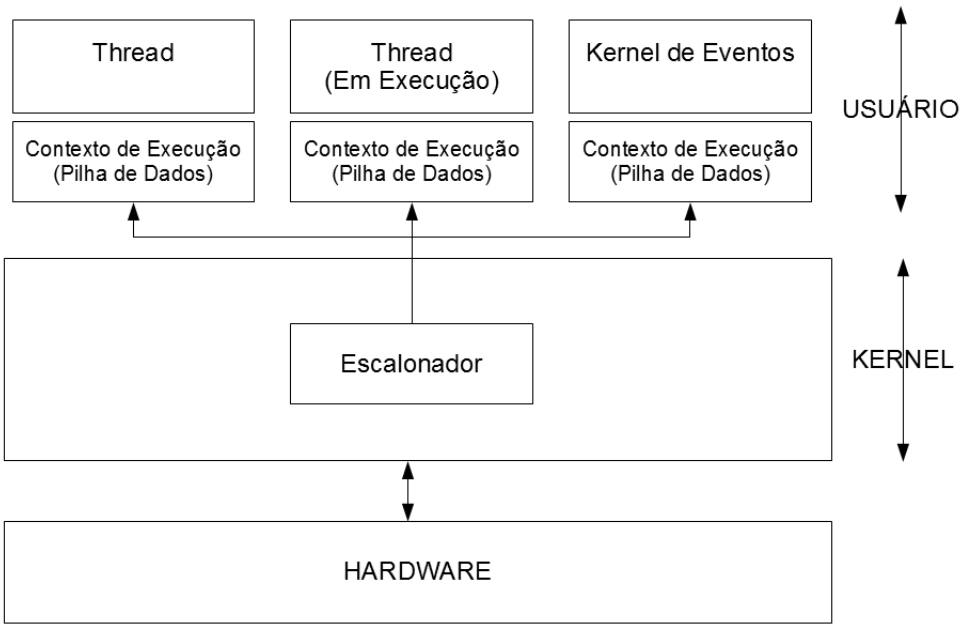

Figura 3.3: estrutura de um sistema baseado em *threads*, mas com suporte a eventos.

3.5 Sistemas Operacionais

Ao longo dos anos, diversas propostas de SOs para RSSFs surgiram para facilitar a criação de aplicações. Dentre essas soluções podem-se destacar: TinyOS (HILL et al., 2000); Contiki (DUNKELS; GRONVALL; VOIGT, 2004); Mantis (ABRACH et al., 2003); HybridKernel (LAUKKARINEN et al., 2009) e algumas extensões do TinyOS, como: TOSThreads (KLUES et al., 2009); TinyThreads (MCCARTNEY; SRIDHAR, 2006) e TinyMOS (TRUMPLER; HAN, 2006).

3.5.1 TinyOS

O TinyOS é um sistema operacional especificamente desenvolvido para sistemas embarcados de rede. O modelo de programação deste SO é voltado para aplicações dirigidas a eventos como também de pequeno tamanho. Os aplicativos para TinyOS são escritos em uma extensão da linguagem C, chamada nesC, que incorpora um modelo de programação orientado a eventos, modelo de concorrência flexível e orientação a componentes (GAY et al., 2003).

Os componentes na linguagem nesC consistem de interfaces na forma de comandos e eventos. As aplicações são formadas através da ligação das interfaces de um conjunto de componentes. A arquitetura de componentes resultante facilita o processamento baseado em eventos, o qual é implementado pelas tarefas e eventos (HILL et al., 2000).

As tarefas e os eventos são os dois tipos de unidades executáveis no TinyOS. Tarefas são mecanismos de computação deferida. Este nome se deve ao fato de que as tarefas executam até o término e não podem ser preemptadas umas pelas outras. Neste SO, componentes podem postar tarefas, sendo que a operação de postar retorna imediatamente, deferindo a computação até que o escalonador execute a tarefa posteriomente. Para assegurar baixa latência de execução, as tarefas devem ser curtas e operações longas devem ser divididas em várias tarefas menores. Da mesa forma que as tarefas, os eventos também executam até o término, mas podem interromper a execução de outro evento ou tarefa. Um evento pode significar a finalização de uma operação *split phase* ou um evento do ambiente.

O paradigma empregado no TinyOS cria um sistema compacto e de baixo consumo de energia, mas torna a programação mais complexa. Os programadores devem dividir atividades conceituais em diversas tarefas pequenas e o fluxo de execução precisa ser controlado por máquinas de estados e variáveis globais.

3.5.2 Contiki

O Contiki é um SO para sistemas com recursos escassos que provê carga dinâmica de componentes de software. O *kernel* deste SO é dirigido a eventos, mas o sistema suporta uma biblioteca opcional que implementa *multithreading*. Este SO sistema foi escrito em linguagem C e possui porte para vários microcontroladores, incluindo MSP430 da Texas Instruments e AVR da Atmel (DUNKELS; GRONVALL; VOIGT, 2004).

O suporte à atualização do código do sistema em tempo de execução é uma funcionalidade importante, visto que, na maioria das aplicações, é caro fazer a atualização de cada nó no local de instalação. A maior parte dos SOs requerem que uma imagem completa do sistema seja utilizada para fazer a atualização. Este requisito vai de encontro as limitações de energia dos nós sensores. Entretanto, o Contiki permite a carga dinâmica de aplicações e serviços em tempo de execução. Normalmente, uma aplicação individual é bem menor do que um sistema completo e requer menos energia para ser disseminada na rede (DUNKELS; GRONVALL; VOIGT, 2004).

Um sistema Contiki em execução consiste do *kernel*, bibliotecas, carregador de programas e um conjunto de processos, cuja comunicação acontece através do *kernel*. Contudo, este SO não provê uma camada de abstração do *hardware*, ou seja, os *drivers* e aplicações podem acessar diretamente os periféricos, o que pode diminuir a portabilidade das aplicações.

O *kernel* do Contiki consiste de um escalonador de eventos que encaminha eventos para os processos em execução e periodicamente chama os gerenciadores de *polling* dos processos. Toda execução de aplicações é acionada pela ocorrência de eventos ou pelo mecanismo de *polling*, que é responsável por verificar o estado dos periféricos. O gerenciador de *polling* pode ser visto como um evento de alta prioridade que é escalonado entre os eventos assíncronos. Em aplicações que possuem tarefas com computações demoradas, os gerenciadores de *polling* podem ter a execução postergada e causar problemas de tempo de resposta aos eventos do *hardware*.

3.5.3 Mantis

O sistema Mantis utiliza um modelo de operação tradicional baseado em *multithreading* preemptivo. Este sistema provê recursos de reprogramação dinâmica em tempo de execução do sistema inteiro ou de seções da memória de programa. Por causa do modelo *multithreading*, o Mantis precisa manter uma pilha de dados para cada *thread*, bem como mecanismos de sincronização e comunicação (ABRACH et al., 2003).

3.5.4 TinyThreads

TinyThread é uma biblioteca que habilita a programação *multithreading* no sistema TinyOS. As tarefas TinyOS executam em uma *thread* de sistema dedicada e *threads* de aplicação cooperativamente compartilham o processador com a *thread* de sistema. Nesse caso, é exigido que as *threads* cedam voluntariamente o processador e é responsabilidade do programador inserir pontos de cessão no código. No entanto, é possível que uma *thread* de longa duração atrapalhe a execução de *threads* com requisitos de tempo crítico.

As *Threads* no TinyThread seguem a definição tradicional; cada *thread* possui sua própria pilha de dados. O ambiente de desenvolvimento provido por esse SO oferece uma ferramenta que analisa estaticamente a utilização da pilha pelas aplicações. Após a análise, a ferramenta indica a quantidade de memória que deve ser alocada para cada *thread* (MCCARTNEY; SRIDHAR, 2006).

O núcleo do TinyThread é um escalonador do tipo FIFO que executa dentro de uma tarefa do TinyOS. Quando uma tarefa do escalonador é chamada, este varre a lista de *threads* até encontrar uma no estado pronto para execução. Quando uma *thread* T_k pronta é encontrada, o escalonador faz a troca da pilha do TinyOS com a pilha de T_k. A *thread* T_k é executada até a finalização ou até fazer uma chamada bloqueante. Em qualquer um dos casos, a pilha do TinyOs é restaurada em seguida.

Cada vez que o escalonador desse SO é executado, apenas uma *thread* da fila é posta em execução. Se houverem N *threads* no estado pronto, o escalonador deve ser chamado N vezes. Embora o TinyThread permita preempção entre *threads*, esta funcionalidade é desabilitada por padrão. Neste caso, uma única *thread* de longo tempo de execução pode atrasar a execução do escalonador do TinyOS e portanto interferir nos serviços críticos do *kernel*.

3.5.5 TOSThreads

TOSThreads é um pacote que adiciona preempção ao TinyOS, mantendo a compatibilidade com as bibliotecas tradicionais do TinyOS. O TOSThreads lida com o compromisso entre a facilidade da programação baseada em *threads* e a eficiência da programação baseada em eventos. Isto é realizado através da execução do código baseado em eventos em uma *thread* de *kernel* de alta prioridade e todo código de aplicação é executado em *threads* de aplicação, que só executam quando o *kernel* está ocioso. Assim, toda a comunicação entre código de aplicação e *kernel* é feita através de troca de mensagens (KLUES et al., 2009).

O escalonador padrão do TOSThreads implementa o algoritmo preemptivo *round-robin* com fatia de tempo de 5 ms, sendo o primeiro componente a tomar controle do processador na sequência de inicialização do sistema, tendo como tarefa encapsular o TinyOS em uma *thread* e acionar a sequência padrão de inicialização do TinyOS. Após a inicialização do TinyOS, o controle é retomado pelo escalonador que inicia o escalonamento das *threads*.

TOSThreads implementa chamadas de sistema bloqueantes através do encapsulamento de serviços orientados a eventos do TinyOS em uma biblioteca de aplicação. Quando uma chamada de sistema é realizada, uma tarefa TinyOS é postada, acionando imediatamente a *thread* que executa o TinyOS. Como o código TinyOS tem prioridade sobre as outras *threads*, o retorno da chamada de sistema é executado antes que seja possível uma nova *thread* inicializar uma nova chamada de sistema. Desta forma, apenas uma tarefa TinyOS

é necessária para executar todas as chamadas de sistema. O código desta tarefa simplesmente chama a função apontada pela mensagem passada pela *thread* solicitante (KLUES et al., 2009).

3.5.6 TinyMOS

O TinyMOS executa uma versão adaptada do TinyOS em uma *thread* dedicada do sistema MANTIS. Esta abordagem provê duas funcionalidades críticas ao TinyOS: escalonamento baseado em prioridades e a capacidade de criar novas *threads*. Estas *threads* podem ser usadas para executar as computações de longa duração e realizar chamadas bloqueantes de entrada e saída. Contudo, o TinyMOS utiliza a técnica de *locking* no *kernel* do TinyOS, apenas uma *thread* por vez pode executar o código do *kernel*. De outra forma, o TOSThreads elimina essa necessidade através da comunicação baseada em troca de mensagens (TRUMPLER; HAN, 2006).

3.5.7 HybridKernel

O HybridKernel apresenta um *kernel* preemptivo combinado com a biblioteca Protothreads que permite a criação de *threads* cooperativas. No HybridKernel, as *threads* preemptivas são chamadas de processos e as protothreads são chamadas apenas de *threads*. Os processos são escalonados por um escalonador preemptivo baseado em prioridades. Assim, o processo com maior prioridade é executado até realizar uma chamada de sistema bloqueante. Cada processo possui pelo menos uma *thread* e as *threads* são conectadas ao seu processo pai (LAUKKARINEN et al., 2009).

O escalonador de *threads* é responsável por chamar cada *thread* pertencente a um processo. Cada *thread* possui um estado, que indica se está bloqueada, suspensa ou pronta para execução. Quando não há nenhuma thread no estado pronto, o escalonador de *threads* suspende o processo corrente. Entretanto, as *threads* não são necessariamente associadas ao tratamento de eventos do sistema, podendo ser utilizadas para outras atividades. Desta forma, caso uma *thread* do processo faça uma chamada de sistema bloqueante, todo o processo e as *threads* associadas ficam sem executar até que a chamada seja atendida.

3.5.8 Análise Comparativa

A Tabela 3.1 apresenta uma síntese das principais características dos SOs apresentados nesta seção. É importante ressaltar que embora os sistemas TinyThreads, TOSThreads

e TinyMOS sejam bibliotecas dos SOs TinyOS e Mantis, nesse estudo são consideradas como um sistema a parte.

	TinyOs	Contiki	Mantis	TinyThreads	TOSThreads	TinyMOS	HybridKernel
Publicação	2000	2004	2005	2006	2009	2006	2009
Concorrência	eventos	threads/eventos	threads	threads/eventos	threads/eventos	threads/eventos	threads/eventos
Linguagem	nesC	C	C	nesC	nesC	C/nesC	C
Reprogramação	Sim	Sim	Sim	Sim	Sim	Sim	Não
Tempo Real Crítico	Não	Não	Não	Não	Não	Não	Não
Tempo Real Suave	Não	Não	Sim	Sim	Sim	Sim	Não
Estrutura	Monolítico	Modular	Modular	Monolítico	Monolítico	Monolítico	Monolítico

Tabela 3.1: tabela comparativa dos SOs

3.6 Conclusão

Os modelos de concorrência de SOs para RSSFs apresentam diversas deficiências que causam impacto no trabalho dos programadores. O modelo baseado em eventos torna difícil o controle dos fluxos de execução e não se ajusta a problemas com longos períodos de computação. Já o modelo baseado em *threads* tem alta ocupação de memória.

Um SO que visa incorporar um modelo de concorrência baseado em *threads* que seja capaz de garantir o atendimento das restrições temporais de *threads* de alta prioridade, ao mesmo tempo que permite a execução de tarefas com longos períodos de computação, sem causar grandes impactos na ocupação de memória de dados e do processador.

Capítulo 4

SKMotes

Ne ste Capítulo são apresentadas as características do SO SKMotes e uma avaliação comparativa de desempenho deste com outros SOs disponíveis no mercado. Na seção 4.1, apresentam-se as principais motivações para o projeto do SKMotes. Logo em seguida, a seção 4.2 examina suas principais características. A arquitetura do *kernel* é apresentada na seção seguinte. A seção 4.6 apresenta os resultados da avaliação de desempenho. Por fim, na última seção, é apresentada a conclusão deste Capítulo.

4.1 Introdução

Um novo SO para RSSFs que explore as facilidades de programação do modelo *threads*, mas com baixo consumo de memória, pode ser uma estratégia adequada e abrangente para o desenvolvimento de novos projetos. Assim, o SKMotes foi projetado com base nessa estratégia. Neste novo SO, o modelo de concorrência proposto é baseado em *threads*, mas não completamente preemptivo, pois em dado momento apenas um subconjunto das *threads* do sistema devem executar no modo preemptivo baseado em prioridades. O restante das *threads* permanece em espera, ocupando apenas um contexto mínimo de execução, que não contempla a pilha de dados. Isto é alcançado através da utilização de chamadas de sistema especiais que bloqueiam as *threads* que esperam por eventos específicos, de forma que o contexto de execução seja apenas o ponto de continuação na rotina principal da *thread*. Assim, o principal objetivo desse modelo é prover tempos de resposta baixos para *threads* de alta prioridade, ao mesmo tempo que garante baixo consumo de energia e ocupação de memória mais baixa que o modelo preemptivo baseado em *threads*.

4.2 Visão Geral

SKMotes é um SO para RSSFs que emprega elementos dos modelos de concorrência baseado em eventos e baseado em *threads*, com o intuito de fornecer um modelo de programação no estilo *thread*, mas com baixo consumo de memória e boa gestão de energia. A denominação SKMotes é uma forma reduzida de *"Small Kernel for Motes"*. A palavra *Motes* é um termo normalmente utilizado para designar os dispositivos sensores.

O projeto do sistema prezou por criar uma arquitetura compacta, portável e fácil de programar. O SKMotes é escrito em linguagem C e os detalhes da plataforma alvo são escondidos pela camada de abstração de *hardware*, que prover a estrutura para execução de um *kernel* simples que implementa um escalonador em dois níveis. O escalonador de primeiro nível gerencia as *threads* organizadas em uma fila com prioridades. As *threads* no primeiro nível possuem contexto mínimo de execução, composto apenas pelo endereço de retorno na função principal da *thread*. Desta forma, o escalonador decide qual *thread* deve ser designada para um contexto de execução completo, composto de pilha de dados e registradores do processador.

No SKMotes, os critérios de alocação das *threads* nos contextos de execução são disponibilidade de contexto, prioridade e categoria da *thread*. As categorias podem ser determinadas pelo programador, como uma forma de reservar contextos para um determinado grupo de *threads*. Por exemplo, as *threads* que realizam tarefas com longos períodos de computação podem ser agrupadas e executarem todas no mesmo contexto. Além disso, as *threads* de sistema, que cuidam da continuidade do tratamento das interrupções de *hardware*, possuem alta prioridade podendo ser executadas em qualquer contexto de execução e podem possuir contexto exclusivo para diminuir seu tempo de resposta.

O escalonador de segundo nível do SKMotes implementa um algoritmo preemptivo baseado em prioridades para as *threads* alocadas nos contextos de execução. Portanto, quando uma *thread* realiza uma operação de E/S, ela perde seu contexto de execução e é adicionada em uma fila de espera. Após a ocorrência do evento esperado, a *thread* é adicionada novamente na fila de *threads* do escalonador de primeiro nível. Esse modelo de concorência foi denominado de semipreemptivo, por possuir, em um dado momento, características do modelo preemptivo apenas para um subconjunto de *threads*.

Usualmente, dois ou três contextos de execução são utilizados no segundo nível. Como dito anteriormente, pode-se reservar um dos contexto de execução para *threads* do sistema ou criar classes de *threads* que compartilham sempre o mesmo contexto. O programador pode ajustar as configurações de contextos de execução e categorias de *threads*, adequando

o SKMotes às especificações e às restrições do conjunto de tarefas da aplicação específica. Além disso, é possível implementar um módulo do *kernel* que proveja o controle dinâmico da alocação dos contextos, tornando o sistema adaptativo às demandas de execução de *threads*. Por exemplo, o sistema, ao detectar um aumento da latência das *threads* com restrições de tempo, pode aumentar o número de contextos alocados para esta categoria de *threads*.

No SKMotes, apenas *threads* pertecentes ao segundo nível podem ser executadas pelo processador. Uma *thread* de segundo nível pode ser preemptada por outra *thread* de maior prioridade ou bloquear esperando por eventos de E/S. Neste caso específico, a *thread* é enviada de volta ao primeiro nível. Quando não há nenhuma *thread* no estado pronto, este sistema é colocado no estado de baixo consumo de energia.

O SKMotes também provê bibliotecas para comunicação e sincronização entre *threads*. Para tanto, o mecanismo utilizado é baseado em *mutexes* e semáforos com herança de prioridades. Além disso, são disponibilizadas bibliotecas para os periféricos mais comuns das plataformas de nós sensores.

4.3 Arquitetura

O SKMotes foi projetado em uma arquitetura estruturada em camadas, conforme ilustrado na Figura 4.1.

Na base da estrutura do SKMotes temos a camada de *Hardware* que consiste de elementos físicos disponibilizados pela plataforma do nó sensor. Como existem diversas plataformas disponíveis, é importante que uma camada de *software* esconda os detalhes dos recursos físicos e proveja uma interface homogênea para diferentes plataformas. Neste sistema, esta função é desempenhada pela camada de Abstração de *Hardware*, que possui as bibliotecas de baixo nível de interface com os periféricos e o processador alvo.

Sobre a camada de Abstração de *Hardware* foi construído o *kernel* do SKMotes. O projeto do *kernel* preza por prover uma arquitetura simples com baixos requisitos de memória. Em síntese, esta camada implementa as funções essenciais do SKMotes, como: escalonamento das *threads*; gerenciamento da E/S e mecanismos de comunicação entre *threads*.

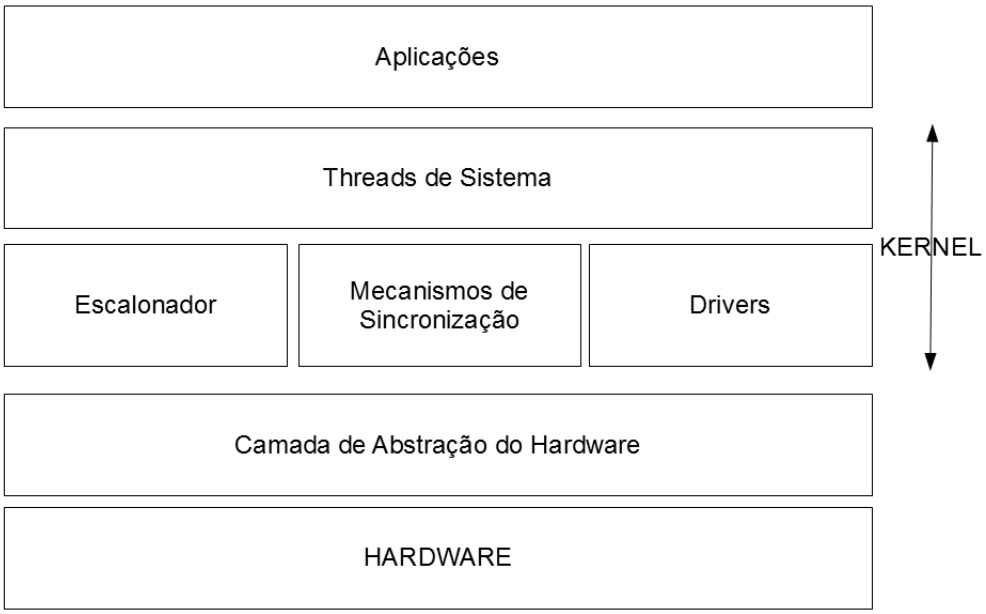

Figura 4.1: estrutura do SKMotes.

4.3.1 Modelo de entrada e saída

O gerenciamento do eventos de E/S do SKMotes é realizado pelos *drivers* de periféricos específicos. Um evento de *hardware* é tratado inicialmente na camada de Abstração de *Hardware*, repassando-o para o *driver* do periférico específico, este por sua vez, pode deferir o gerenciamento deste evento para uma ou mais *threads* de sistema que possuem alta prioridade de execução. A última camada da arquitetura SKMotes consiste das *threads* de aplicação que são criadas pelos programadores do sistema e fazem uso dos serviços fornecidos pelo *kernel*.

4.3.2 Modelo de concorrência

O SKMotes fornece um modelo de concorrência semipreemptivo que é baseado em *threads*, mas com alterações que visam reduzir a demanda de memória utilizada para salvar o contexto de execução de *threads*.

O modelo de concorrência utilizado no SKMotes é ilustrado na Figura 4.2. Quando uma *thread* é criada, esta é submetida ao escalonador de primeiro nível, que a adiciona à lista de *threads* prontas. Neste ponto, o contexto de execução salvo pelo sistema é

mínimo, consistindo apenas do endereço da primeira instrução da função principal da referida *thread*. Por exemplo, no microcontrolador MSP430, o contexto mínimo consiste de apenas dois bytes.

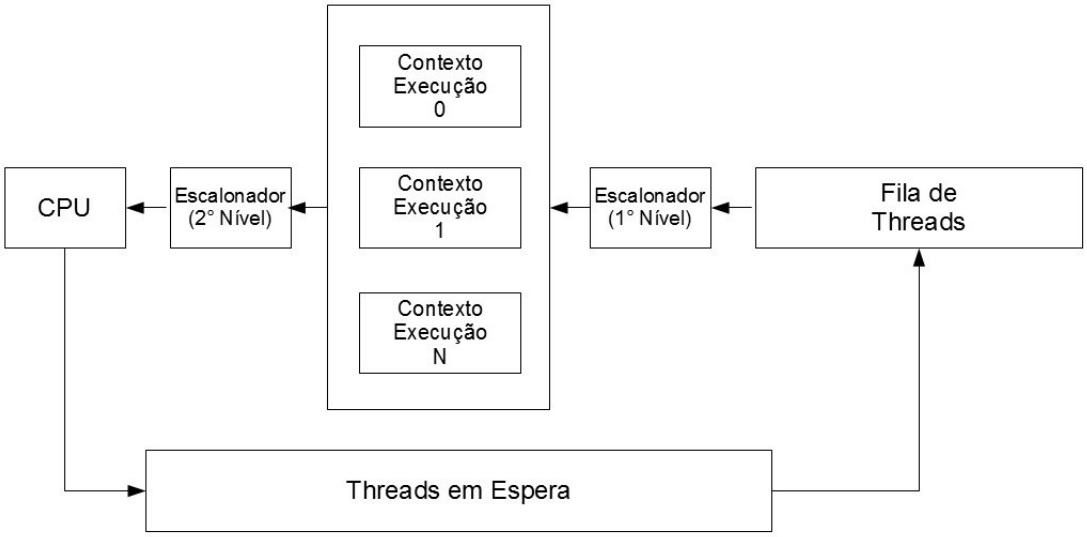

Figura 4.2: diagrama ilustrativo do escalonamento das *threads*.

Uma *thread* pronta, alocada no escalonador de primeiro nível, só é submetida ao escalonador de segundo nível, se sua prioridade for superior às prioridades das *threads* de segundo nível e se houver um contexto de execução disponível. Ao ser submetida ao escalonador de segundo nível, esta *thread* passa a ter um contexto de execução completo com pilha de dados e estado dos registradores do processador. Isto é necessário, pois, no segundo nível é utilizado um algoritmo de escalonamento preemptivo baseado em prioridades. Uma *thread* pode ser retirada do processador de forma assíncrona, caso exista uma *thread* pronta de maior prioridade. Uma *thread* de segundo nível só é retirada desse nível caso termine sua execução ou faça uma solicitação de um serviço de E/S. Neste caso, esta *thread* é adicionada à lista de *bloqueadas* do escalonador de primeiro nível, e só é retirada quando o serviço solicitado for realizado. Assim, o contexto de execução utilizado é liberado e pode então ser utilizado por outra *thread*. Um contexto de execução pode ser reservado para *threads* de sistema, para agilizar o tratamento dos eventos de E/S e do *kernel*. Entretanto, o número de contextos de execução deve ser reduzido para não causar impactos negativos na ocupação da memória de dados. Os estados de uma *thread* no modelo de concorrência do SKMotes são mostrados na Figura 4.3.

O modelo de concorrência do SKMotes não implementa um sistema preemptivo por completo. Assim, em dado instante, apenas um subconjunto de *threads* do sistema estam

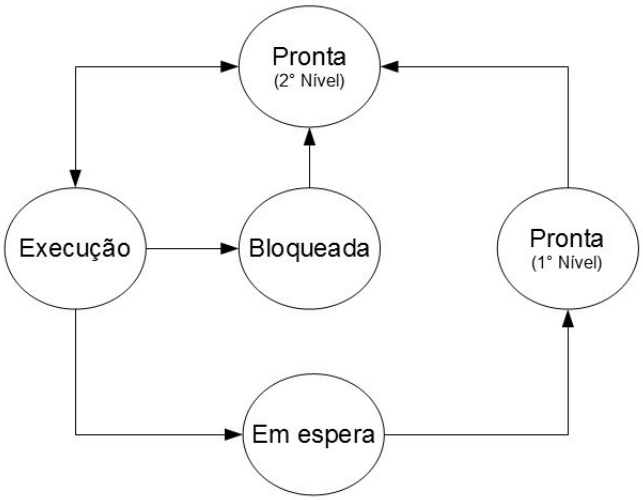

Figura 4.3: máquina de estados do escalonamento das *threads*.

executando no modo preemptivo e as outras *threads* estão em espera com apenas um contexto mínimo de execução que não contempla a pilha de dados e o estado dos registradores. Por isto, o bloqueio de uma *thread* só pode ocorrer em chamadas especiais do *kernel*. Esta implementação adiciona algumas desvantagens ao modelo de programação. As variáveis utilizadas na função principal da *thread* devem ter escopo global ou estático, e os pontos de bloqueio só podem ser executadas da função principal, pois o histórico de chamadas de rotinas, que é armazenado na pilha de dados, não é salvo. Além disso, em algumas situações, *threads* de alta prioridade podem ser forçadas a esperar por um contexto de execução, o que adiciona uma latência variável na execução. Contudo, essas limitações são compensadas pelos ganhos na ocupação da memória e estilo de programação.

4.3.2.1 Exemplo do funcionamento

Para ilustrar o funcionamento do modelo de concorrência semipreemptivo, apresenta-se um caso de uso de um sistema que possui três contextos de execução e, inicialmente, quatro *threads*. Estes contextos de execução são divididos para duas categorias de *threads*: alta e baixa prioridade. Dois dos contextos são alocados para *threads* de alta prioridade, o restante é alocado para *threads* de baixa prioridade. A Figura 4.4 mostra o estado inicial do sistema, os círculos representam as *threads*, o valor escrito nestes a prioridade e identificação destas *threads*. Neste exemplo, *threads* com prioridade igual ou superior a cinco são consideradas de alta prioridade, e possuem os contextos 0 e 1 reservados para sua execução.

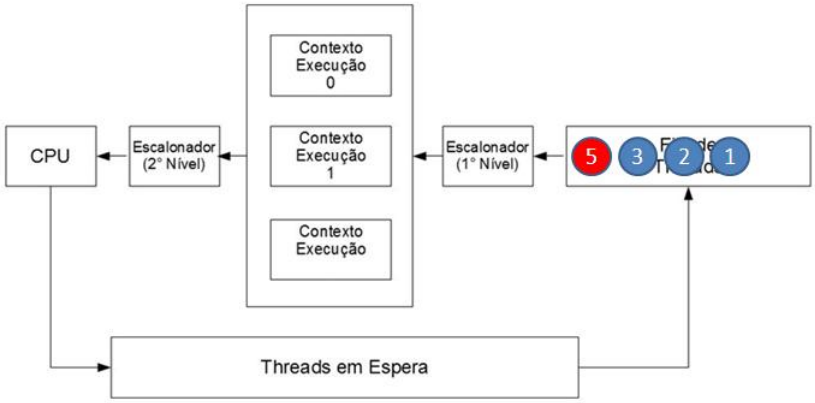

Figura 4.4: estado inicial do sistema.

O escalonador de primeiro nível aloca as *threads* da fila de *threads* prontas nos contextos disponíveis, verificando a prioridade e categoria das *threads*. A Figura 4.5 mostra o estado após a alocação. A *thread* de alta prioridade de identificador cinco é alocada no contexto de execução 0, enquanto a *thread* de baixa prioridade de identificador três é alocada no último contexto de execução, que é reservado para as *threads* de baixa prioridade. O contexto de execução 1 ficou sem ser utilizado, pois não há mais nenhuma *threads* de alta prioridade na fila de *threads*.

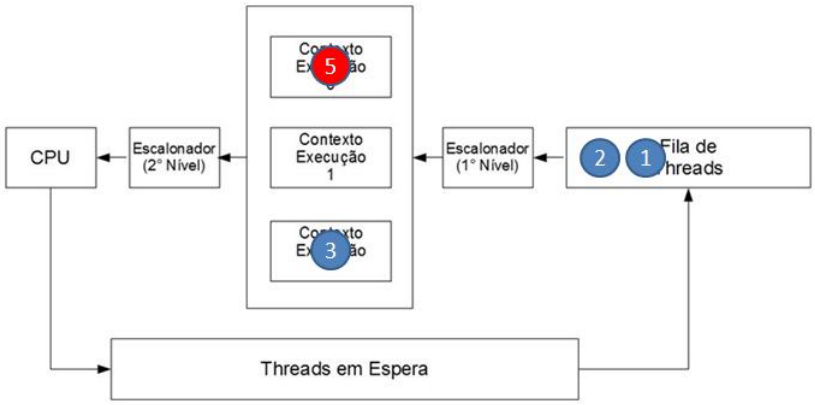

Figura 4.5: alocação nos contextos de execução.

Em seguida, o escalonador de segundo nível utiliza o algoritmo preemptivo baseado em prioridades para definir qual será a próxima *thread* que executará, a escolha é limitada as *threads* alocadas em contextos de execução. Nesse caso, a *thread* de identificador cinco é escolhida. Essa ação é mostrada na Figura 4.6.

Dessa forma, a *thread* de identificador cinco, durante sua execução, cria uma nova

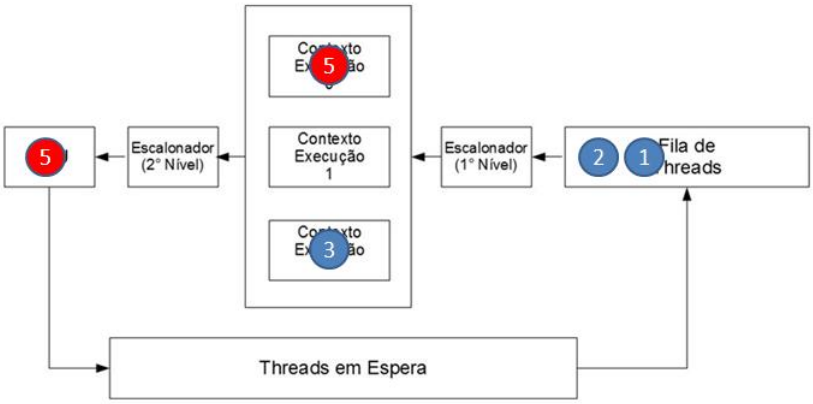

Figura 4.6: escalonamento da *thread* de identificador cinco.

thread, que será adicionada na fila de *threads* do escalonador de primeiro nível, como mostrado na Figura 4.7. No caso, a *thread* recém criada possui prioridade seis. Assim, o escalonador de primeiro nível a aloca no contexto de execução que estava disponível, reservado para *threads* de alta prioridade, como apresentado na Figura 4.8.

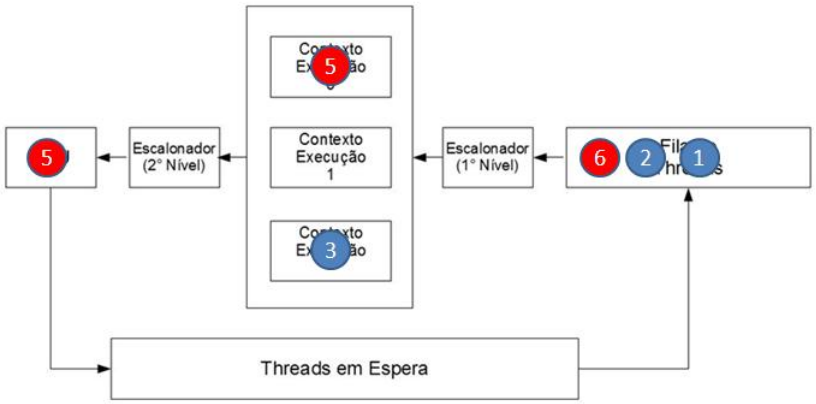

Figura 4.7: criação da *thread* de identificador seis.

Após a alocação da última *thread*, o escalonador de segundo nível percebe que há uma *thread*, alocada em um contexto, com prioridade maior do que *thread* em execução, dessa forma é realizada o salvamento do contexto de execução desta última. Em seguida, a *thread* de identificador seis é posta em execução. O estado do sistema após essa ação é mostrado na Figura 4.9.

A *thread* em execução faz uma solicitação a um serviço de E/S, o escalonador de segundo nível aloca esta *thread* na fila de *threads* em espera, salvando o endereço de retorno da chamada de sistema do serviço de E/S. Contudo, o contexto de execução que pertencia a

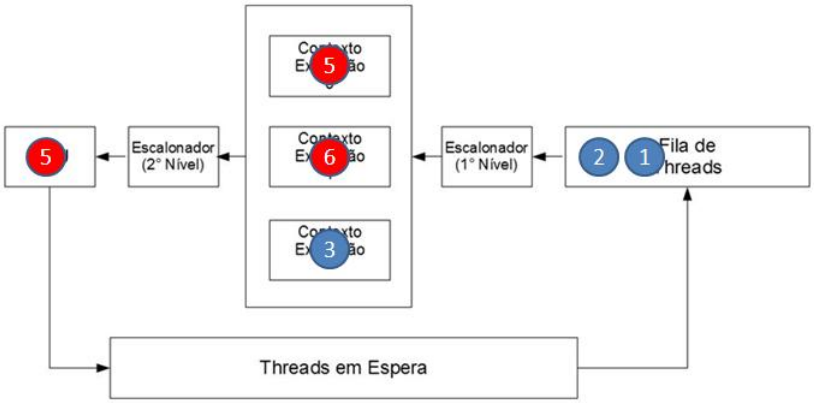

Figura 4.8: alocação da *thread* de identificador seis no contexto de execução.

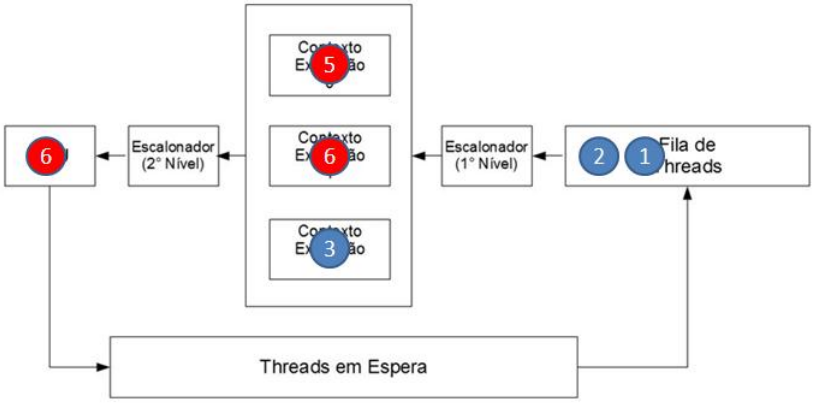

Figura 4.9: preempção da *thread* de identificador cinco.

essa *thread* é posto como disponível, para que possa ser usado por outras *threads*, enquanto a *thread*, que estava em execução, espera pela resposta do serviço solicitado. Em seguida, o processador é entregue a *thread* pronta de maior prioridade. A Figura 4.10 ilustra este estado do sistema. Quando o serviço de E/S é realizado, a *thread*, que o solicitou, é posta de volta na fila de *threads* do escalonador de primeiro nível, como mostrado na Figura 4.11.

4.4 Exemplo de aplicação

Para ilustrar o modelo de programação do SKMotes, apresenta-se um pequeno exemplo de aplicação adaptado de Gay et al (GAY et al., 2003). A aplicação consiste de um componente que periodicamente faz uma leitura de um conversor analógico-digital e a

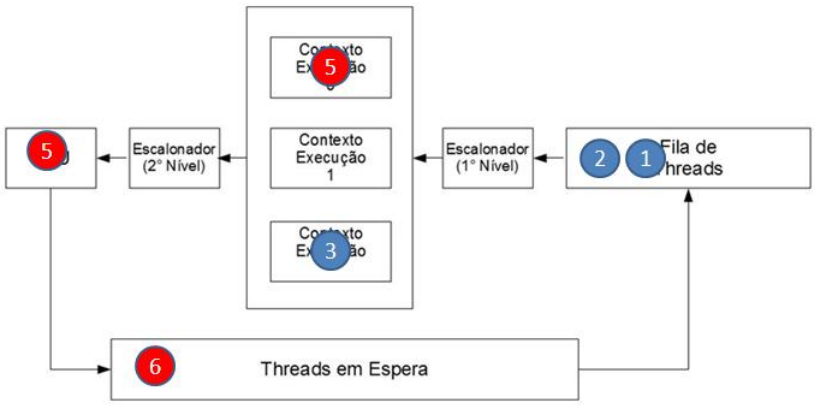

Figura 4.10: solicitação de serviço de E/S da *thread* de identificador seis.

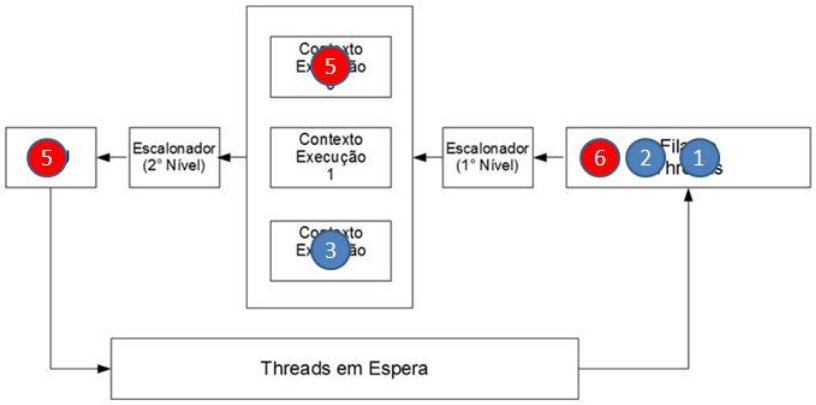

Figura 4.11: término do serviço de E/S.

envia pela rede, cujo código se encontra a seguir.

```
uint16_t sensorReading;

_thread surgeM(void){

  while(1){
    waitTimer(TIME);
    readADC(&sensorReading);
    p.data = sensorReading;
    sendPacket(p);
  }
}
```

Apenas uma *thread* é suficiente para construir a aplicação. A função *waitTimer* deixa a *thread* no modo de espera por um período determinado no parâmetro de entrada da função. Logo em seguida, a leitura do conversor analógico-digital é realizada pela função *readADC* e, por fim, o pacote é enviado para a rede pela função *sendPacket*. Nesse exemplo, todas as operações de E/S são síncronas. Quando uma *thread* solicita a operação de E/S, esta *thread* é posto no estado de espera, até que o serviço seja terminado. Contudo, para o programador o fluxo de execução é contínuo, diferente do modelo de programação apresentado pelos *kernels* baseados em eventos.

Para fazer um paralelo com o modelo de programação utilizado em *kernels* baseados em eventos, segue abaixo o código da mesma aplicação para o TinyOS.

```
Module SurgeM{...}
implementation{
  norace uint16_t sensorReading;

  event result_t Timer.fired(){
    call ADC.getData();
    return SUCESS;
  }

  event result_t ADC.getReady(uint16_t data){
    sensorReading = data;
    post sendData();
    return SUCESS;
  }

  task void sendData(){
    adcPacket.data = sensorReading;
    call Send.send(&adcPacket,sizeof adcPacket.data);
    return SUCESS;
  }

}
```

A função *Timer.fired* é responsável por tratar um evento do temporizador. O comando *ADC.getData* é chamado durante a execução de *Timer.fired*. Contudo, a resposta para a solicitação de leitura do A/D é retornada em outro tratador de eventos: a rotina *ADC.getReady*. Este tratador de evento posta a tarefa *sendData*, ou seja, a insere na FIFO do TinyOS. Após ser selecionada para execução pelo escalonador, esta tarefa pode enviar o pacote para a rede.

Em sistemas baseados em eventos, um código da tarefa conceitual é espalhado em diversos tratadores de eventos e tarefas, diminuindo o entendimento do código em aplicações mais complexas. O SKMotes apresenta um modelo de programação mais simples, pois dá ao programador a possibilidade de utilizar bibliotecas de E/S síncronas, semelhante ao sistemas baseados em *threads* preemptivos, contudo o modelo do SKMotes apresenta uma menor ocupação da memória de dados.

4.5 Análise Comparativa

A Tabela 4.1 reapresenta uma síntese das principais características dos SOs descritos no capítulo anterior, com a adição das informações do SKMotes, que é o único que emprega o modelo de concorrência semipreemptivo que permite ocupação de memória de dados abaixo do modelo preemptivo baseado em *threads* e tempos de resposta inferiores aos *kernels* baseados em eventos.

	TinyOs	Contiki	Mantis	TinyThreads	TOSThreads	TinyMOS	HybridKernel	**SKMotes**
Publicação	2000	2004	2005	2006	2009	2006	2009	2011
Concorrência	eventos	threads-eventos	threads	threads-eventos	threads-eventos	threads-eventos	threads-eventos	semi-preemptivo
Linguagem	nesC	C	C	nesC	nesC	C/nesC	C	C
Reprogramação	Sim	Sim	Sim	Sim	Sim	Sim	Não	Não
Tempo Real Crítico	Não	Não	Não	Não	Não	Não	Não	Não
Tempo Real Suave	Não	Não	Sim	Sim	Sim	Sim	Não	Sim
Estrutura	Monolítico	Modular	Modular	Monolítico	Monolítico	Monolítico	Monolítico	Monolítico

Tabela 4.1: tabela comparativa dos SOs

4.6 Avaliação de desempenho

A avaliação do desempenho do SKMotes foi realizada na plataforma de aplicações embarcadas em rede eZ430-RF2500 (TEXAS INSTRUMENTS, 2011), baseada no microcontrolador MSP430 da Texas Instruments. Esta avaliação consiste em analisar o desempenho do SKMotes em cinco diferentes casos de teste, que refletem cenários típicos de aplicações de RSSFs. As métricas utilizadas nesta avaliação para comparação do desempenho são: ocupação da memória; tempo de resposta das *threads* e utilização da CPU, que é um indicativo do consumo de energia. Para cada caso de teste são realizadas 33 repetições e calculados os valores médios, que são acompanhados do intervalo de confiança de 95%. Note que há uma grande variação dos valores de tempo de resposta, visto que os *kernels* de evento não garantem o atendimento dessa métrica.

Além do SKMotes, os sistemas TinyOS, Contiki e uma versão preemptiva do SKMotes são testados com o intuito de comparar a performance de sistemas baseados nos diferentes modelos de concorrência. Para realizar as medições, um ambiente de testes foi projetado baseado em circuito digital, cujo funcionamento é detalhado em seguida.

4.6.1 Ambiente de testes

Para realizar as medições para a avaliação, de forma pouca invasiva e com alta confiabilidade, foi projetado um ambiente de testes utilizando um *kit* de avaliação baseado em uma FPGA da família Spartan 3E fabricado pela Digilentinc (DIGILENTINC, 2011). O modelo estrutural do circuito de testes está descrito na linguagem VHDL e sintetizado para o dispositivo alvo. A Figura 4.12 ilustra a organização do ambiente de testes.

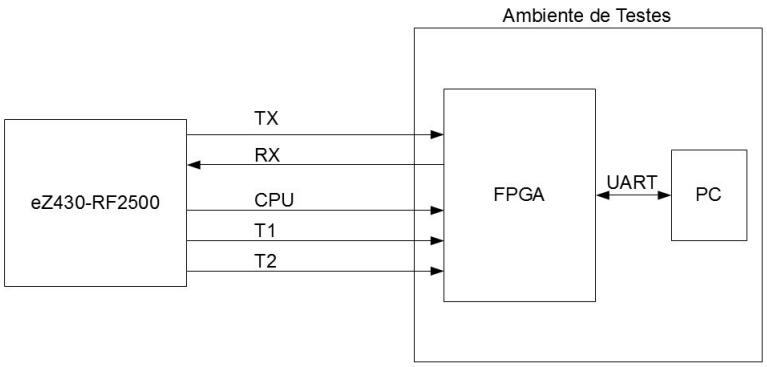

Figura 4.12: estrutura do ambiente de testes.

Cinco sinais são utilizados para fazer a integração do módulo eZ430-RF2500 com o ambiente de testes. Os sinais TX e RX são utilizados para fazer comunicação serial assíncrona, tornando possível que o ambiente de testes envie e receba pacotes por essa interface. O sinal CPU indica se a mesma está ativa ou em modo de baixo consumo, permitindo quantificar a utilização deste recurso. Os sinais T1 e T2 são utilizados para indicar o momento em que as *threads* do sistema realizam suas tarefas, de modo a permitir a medição do tempo de resposta médio destas. Além disso, a FPGA envia as informações coletadas para um computador pessoal utilizando uma interface de comunicação serial.

O *kit* de avaliação utilizado possui fonte de clock de 50 MHz, que é empregada como referência para os circuitos de teste. A Figura 4.13 representa o modelo simplificado do circuito de teste. A descrição funcional da máquina de estados da unidade de controle, escrita em VHDL, pode ser modificada para acomodar as exigências de outros casos de testes.

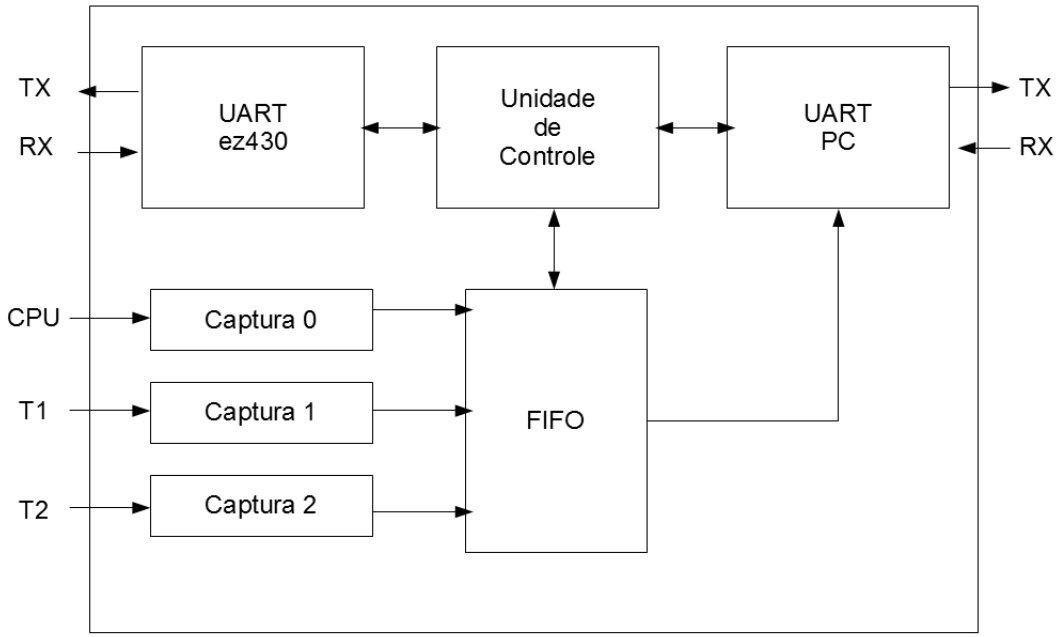

Figura 4.13: estrutura do circuito de teste.

O circuito de teste possui dois módulos de comunicação serial assíncrona, usados para comunicar com o módulo ez430-RF2500 e com o PC, que recebem e transmitem os resultados das medições. Além disso, três módulos de captura são disponibilizados para capturar os valores de um temporizador de 32 bits a cada evento de transição de borda

dos sinais CPU, T1 ou T2. Tais valores são armazenados em um *buffer* do tipo FIFO com capacidade para 256 amostras. Estes valores são posteriormente enviados para o PC através da porta serial. Essas ações são coordenadas pela unidade de controle.

Esse ambiente de testes pode ser reutilizado e/ou estendido para diversos cenários de avaliação de desempenho de sistemas computacionais baseados em dispositivos microcontrolados.

4.6.2 Caso de teste 1

O primeiro caso de teste consiste na execução de uma aplicação que utiliza a interface de comunicação serial para troca de pacotes entre um módulo sensor e uma estação base, denominada de *simpleUART*. A comunicação é iniciada pela estação base que envia um pacote com n bytes. Após a recepção do pacote, o módulo responde com outro pacote de n bytes. O tamanho do pacote varia de 3 a 27 bytes, em incrementos de 6 bytes. Neste caso de teste, a métrica de avaliação foi a ocupação da CPU, cujos resultados são mostrados na Figura 4.14. Os resultados, conforme a Figura 4.14, indicam que o sistema Contiki obtém a menor ocupação da CPU para todos os tamanhos de pacotes. Já o desempenho do TinyOS é ligeiramente inferior. O kernel no modo preemptivo e o SKMotes obtém o pior desempenho, que é explicado pela maior complexidade das rotinas de escalonamento destes *kernels*, em relação às soluções baseadas em eventos.

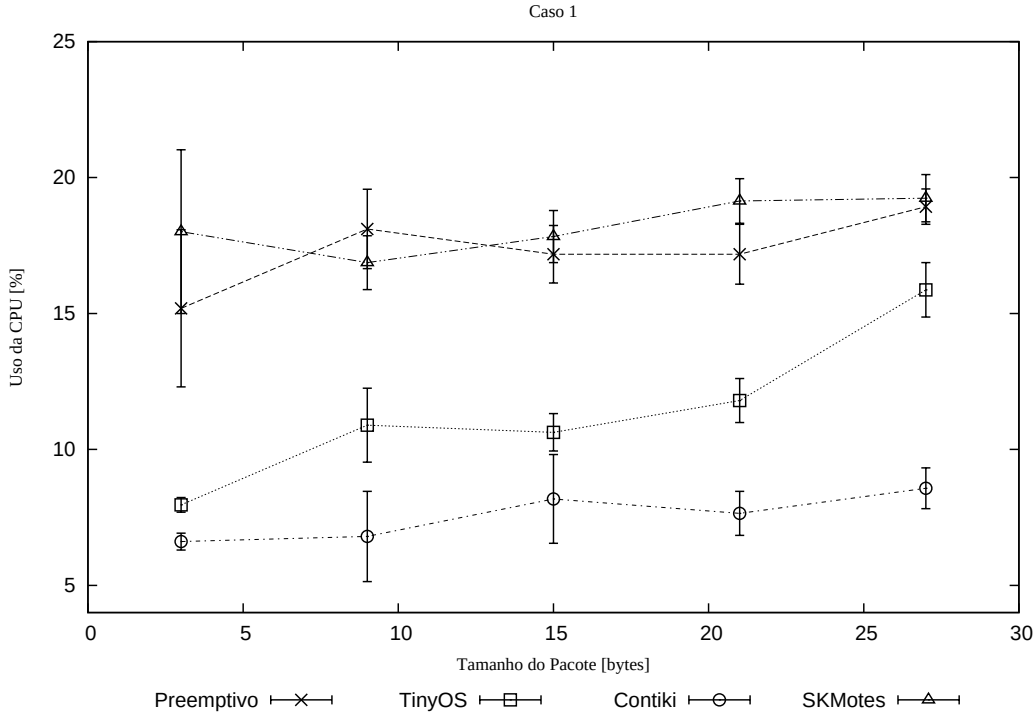

Figura 4.14: uso da CPU - caso de teste 1.

4.6.3 Caso de teste 2

No segundo caso de teste, duas aplicações são executadas. A primeira aplicação é *SimpleUART*, utilizada no caso anterior. A segunda aplicação é chamada de *Blink 1*, que consiste de uma tarefa periódica que aciona um dos pinos de saída de um módulo sensor. As métricas de avaliação são ocupação da CPU e tempo de reposta da *thread* periódica. Neste caso de teste, o SKMotes utiliza apenas um contexto de execução.

A Figura 4.15 mostra os resultados dos testes do caso 2, e mostra que, o sistema Contiki obtêm a menor alocação da CPU, contudo note que para um pacote de tamanho 15 bytes há uma redução no tempo de resposta médio, em relação ao pacote de tamanho 9 bytes, isto é justificado pelo fato do Contiki não garantir tempo de resposta para as tarefas, causando grande variação nos tempos medidos. A solução preemptiva e o SKMotes alcançam desempenho similar. Por outro lado, o TinyOS teve ocupação da CPU maior do que todas as outras soluções.

Em relação ao tempo de resposta, como apresentado na Figura 4.16, a solução preemptiva obtém tempos de respostas com pouca variação para cada cenário deste teste. Entretanto, o Contiki apresenta tempos de respostas superiores ao SKMotes utilizando um único contexto de execução, mas ambos sofrem variação do tempo de resposta com o aumento do tamanho dos pacotes.

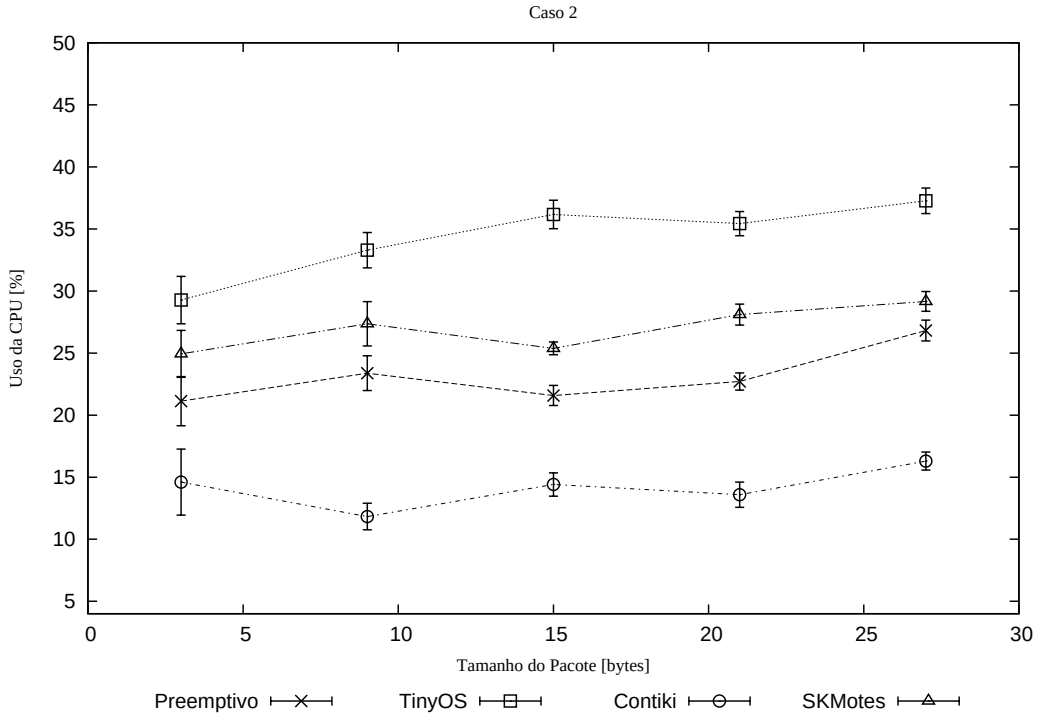

Figura 4.15: uso de CPU - caso de teste 2.

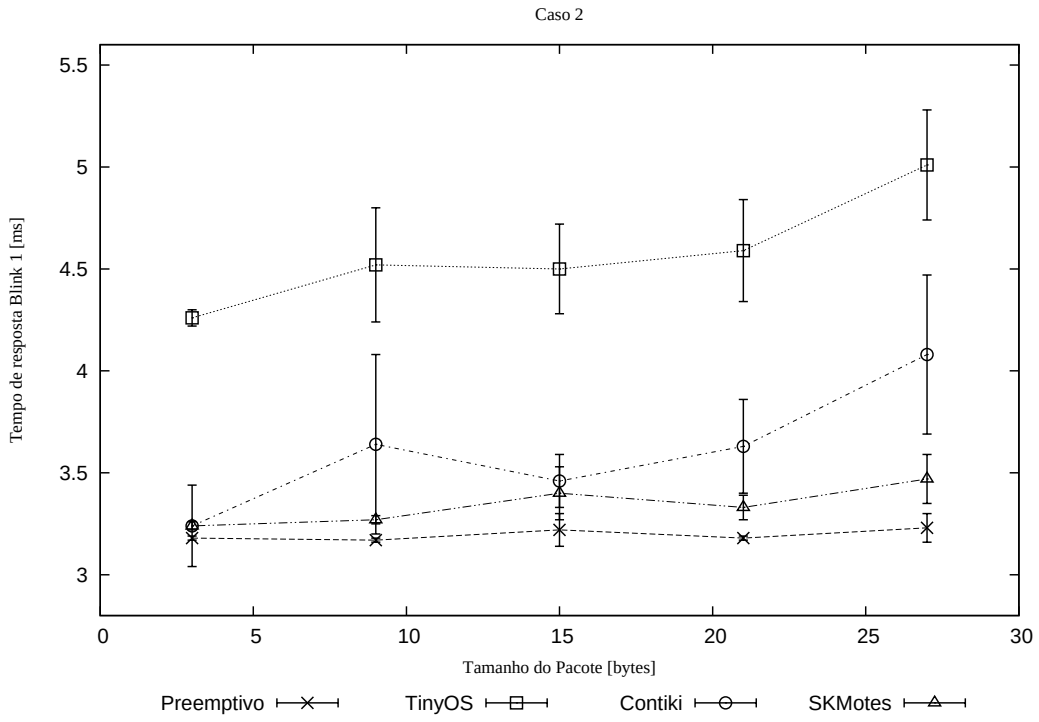

Figura 4.16: tempo de resposta - caso de teste 2.

4.6.4 Caso de teste 3

Neste caso de teste, três aplicações são executadas. Além das duas aplicações do caso de teste anterior, adiciona-se uma outra tarefa com requisitos temporais, denominada *Blink 2*. As métricas de avaliação são novamente a ocupação da CPU e o tempo de resposta médio das tarefas. Neste caso, o SKMotes é testado em duas versões com um e dois contextos de execução, enquanto a solução preemptiva possui um contexto para cada tarefa.

A ocupação da CPU no terceiro caso de teste é apresentada na Figura 4.17. Novamente, o Contiki apresenta a menor ocupação do processador. As versões do SKMotes com um e dois contextos, e a versão preemptiva obtém ocupação da CPU similares, sendo que o TinyOs apresenta a maior ocupação de CPU.

Os valores dos tempos médios de resposta das duas tarefas com requisitos temporais são apresentados nos gráficos das Figuras 4.18 e 4.19. Novamente, o SKMotes com dois contextos e o *kernel* preemptivo obtém variação mínima nos tempos de resposta. Já o TinyOS apresenta variações limitadas no tempo de resposta, sem relação aparente com o tamanho do pacote de dados, mas com valores de tempo de resposta bem acima dos demais. Entretanto, o Contiki e o SKMotes com um contexto de execução apresentam desempenho similar. É importante notar que a adição de um contexto de execução no

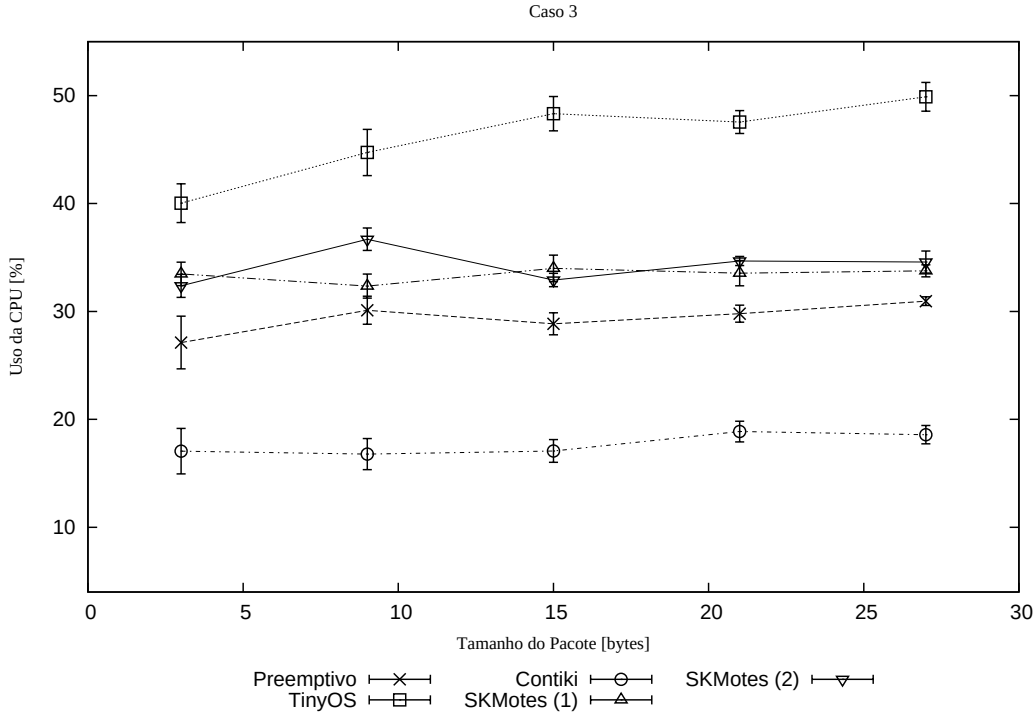

Figura 4.17: uso da CPU - caso de teste 3.

SKMotes reduz a variação nos tempos de resposta das tarefas e a ocupação da CPU, como esperado.

O desempenho de cada sistema em relação a utilização do processador, para os três primeiros casos de teste, é mostrado nas Figuras 4.20, 4.21, 4.22 e 4.23 . Esta métrica reflete em como o sistema faz uso da energia, quanto menor ocupação, melhor a utilização dos recursos de energia. O sistema Contiki apresenta a menor ocupação da CPU, contudo não é capaz de garantir as restrições de tempo das tarefas. Já os sistema preemptivo alcança resultados próximos ao Contiki e garante baixa variação dos tempos de resposta.

Neste teste, o SKMotes, utilizando a solução semipreemptiva, obtém ocupação compatível com as duas outras soluções, mas com valores ligeiramente mais elevados. Já o TinyOS apresenta bons resultados para o primeiro caso de teste, mas ao ser submetido a tarefas com requisitos de temporais, não obtém bons resultados.

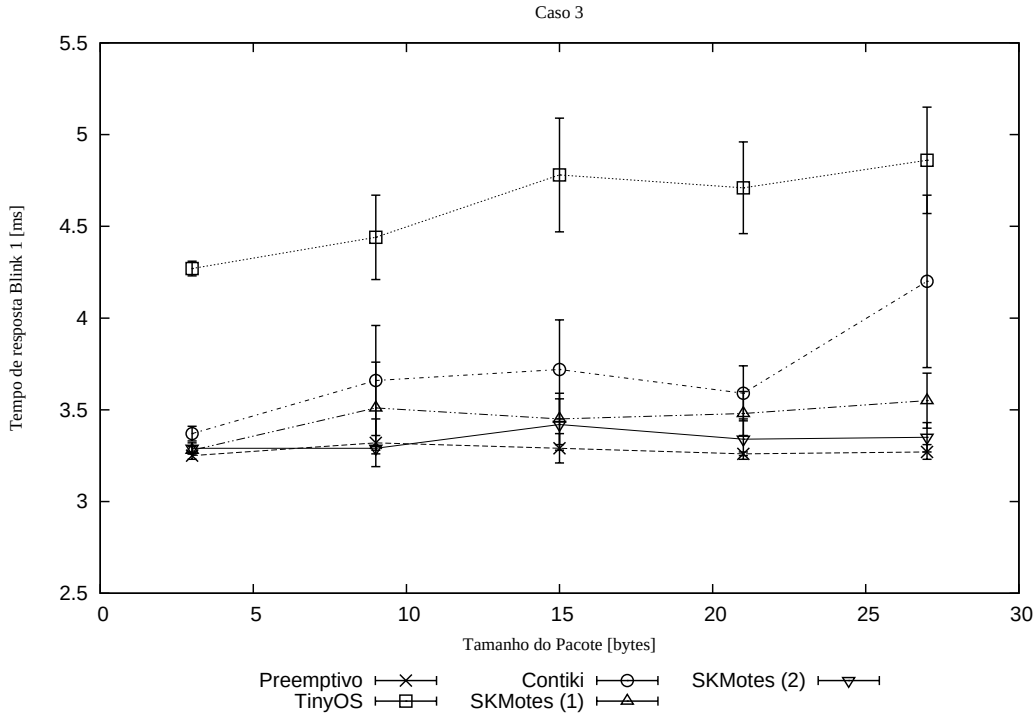

Figura 4.18: tempo de resposta *Blink 1* - caso de teste 3.

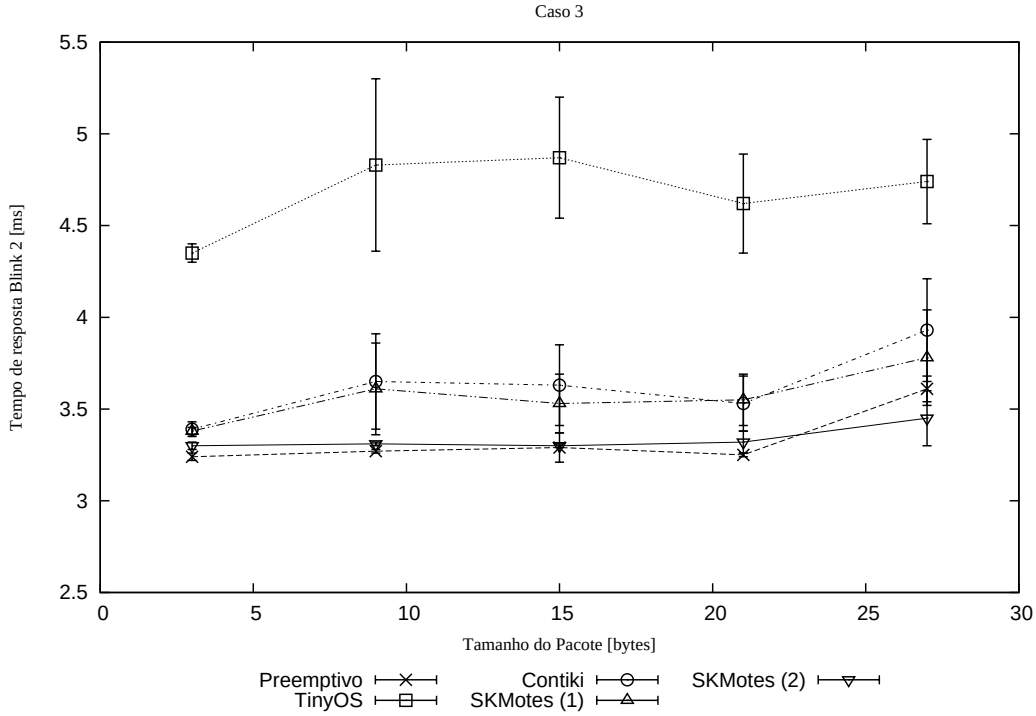

Figura 4.19: tempo de resposta *Blink 2* - caso de teste 3.

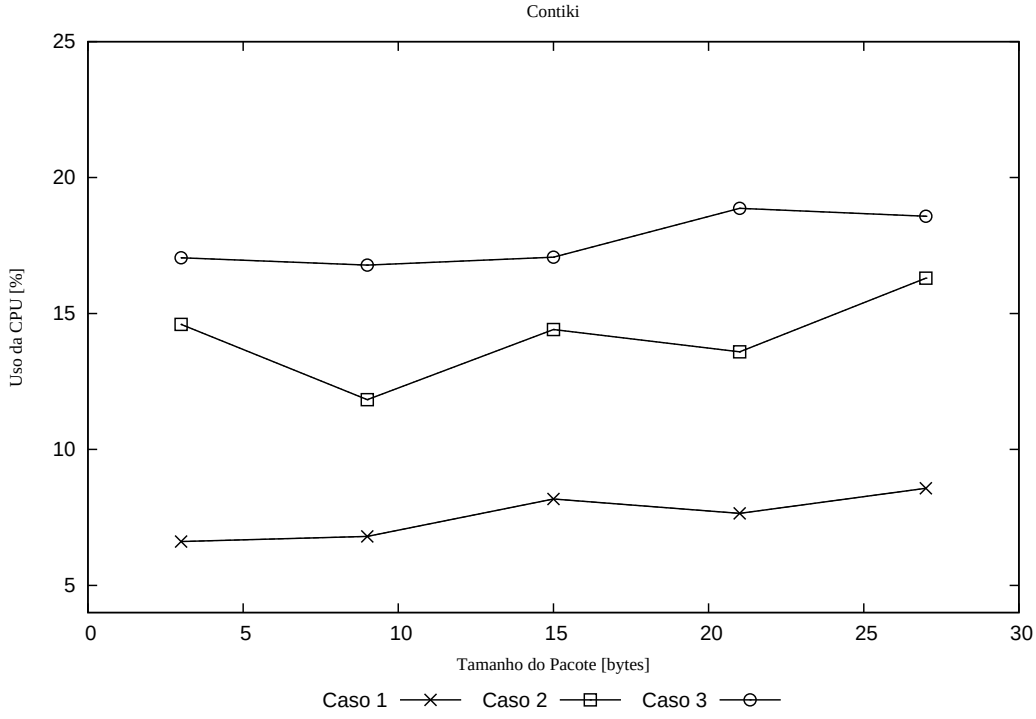

Figura 4.20: uso da CPU - Contiki.

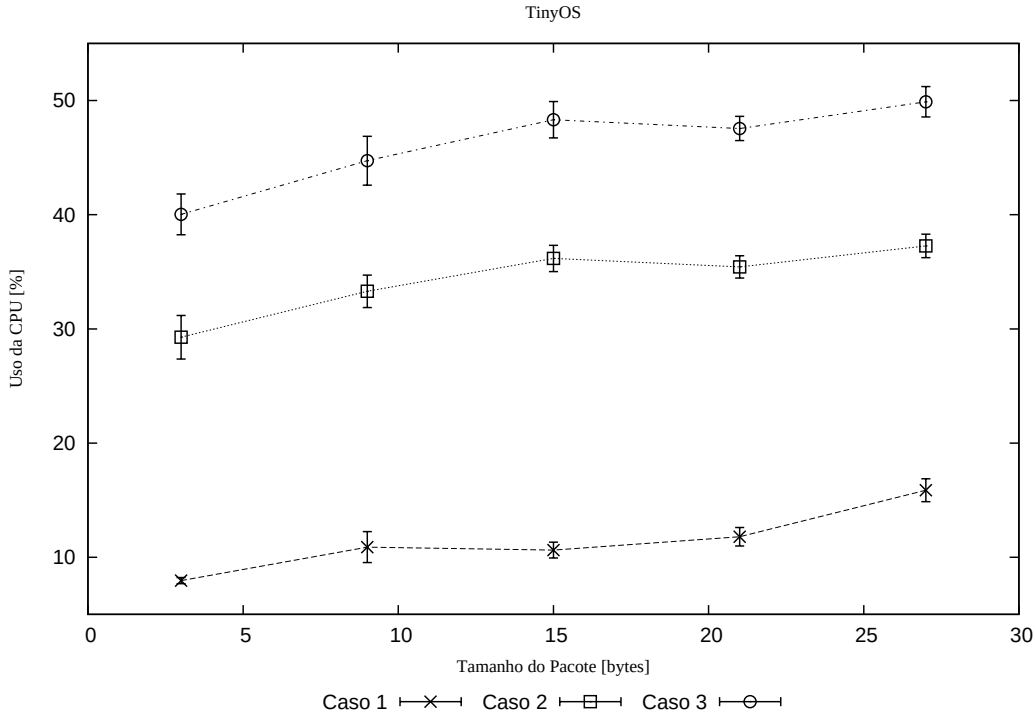

Figura 4.21: uso da CPU - TinyOS.

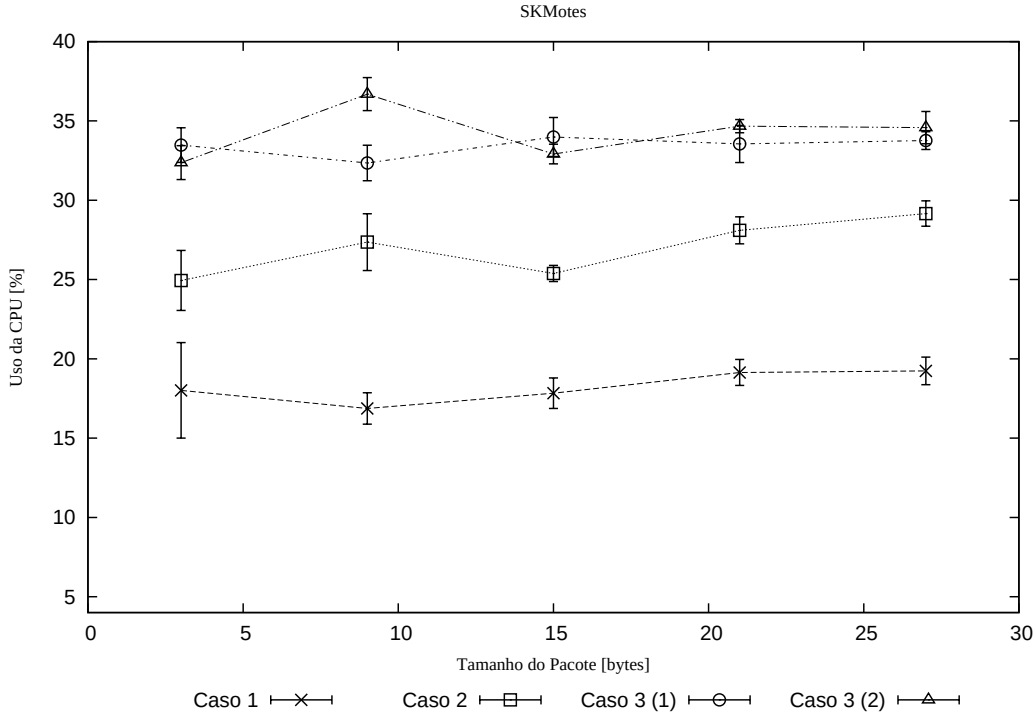

Figura 4.22: uso da CPU - SKMotes.

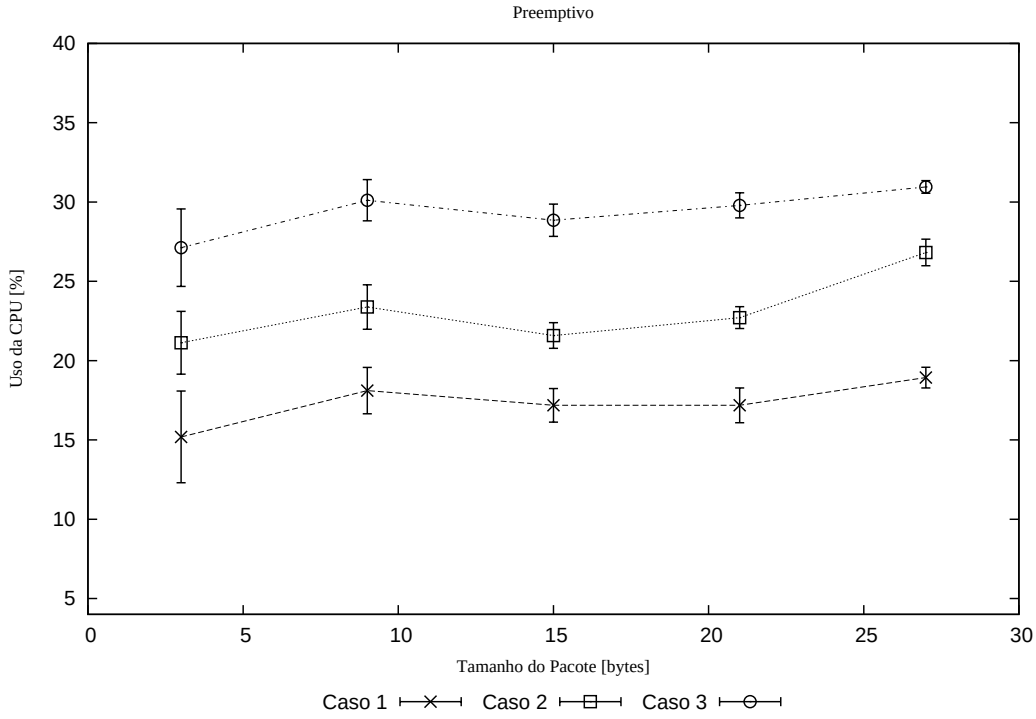

Figura 4.23: uso da CPU - Preemptivo.

4.6.5 Caso de Teste 4

Os três primeiros casos de teste utilizaram tarefas orientadas a E/S, para esse tipo de tarefa, os kernels orientados a eventos tendem a ter desempenho superior. Neste caso de teste, é utilizado um conjunto de tarefas com orientação para operações de E/S e para longos períodos de computação. Este conjunto de tarefas é composto pelas aplicações *Blink 1* e *Blink 2*, além de uma nova tarefa, chamada *longCompute 1* que executa uma tarefa computacional de tempo de execução T, depois permanece 5.0 ms no estado de espera. Este tempo T varia de 3.0 ms até 9.0 ms, com incrementos de 1.0 ms. As métricas de avaliação, para este caso de teste, são ocupação da CPU e tempo de resposta da tarefa *Blink 1*. Neste caso de teste, o SKMotes está configurado para utilizar dois contextos de execução.

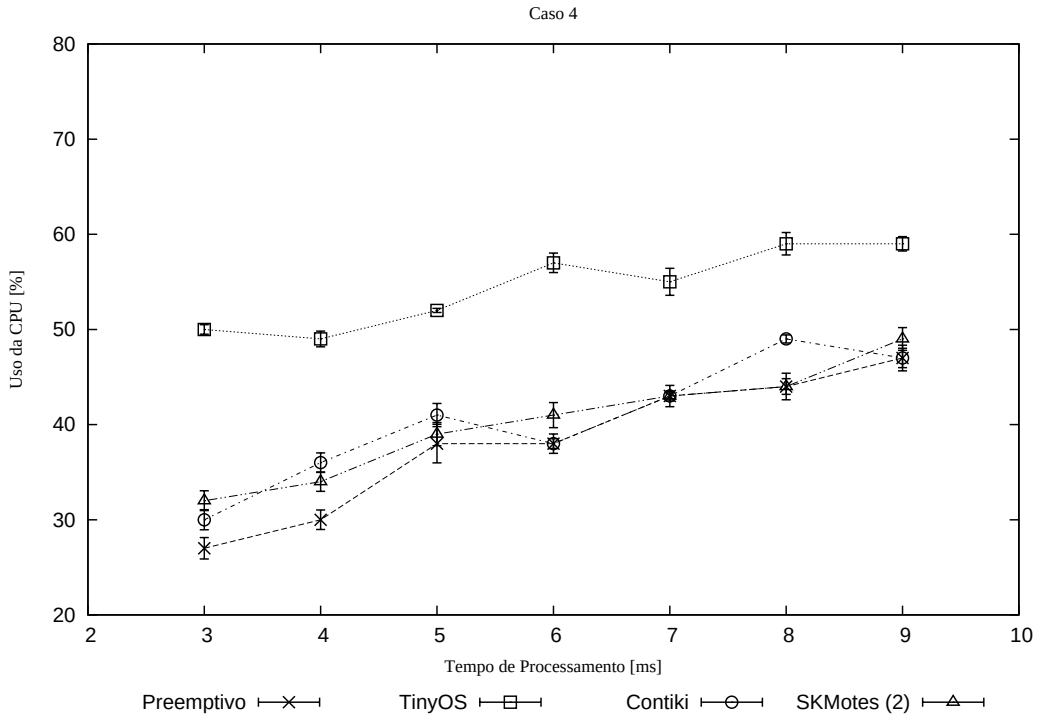

Figura 4.24: uso da CPU - caso de teste 4.

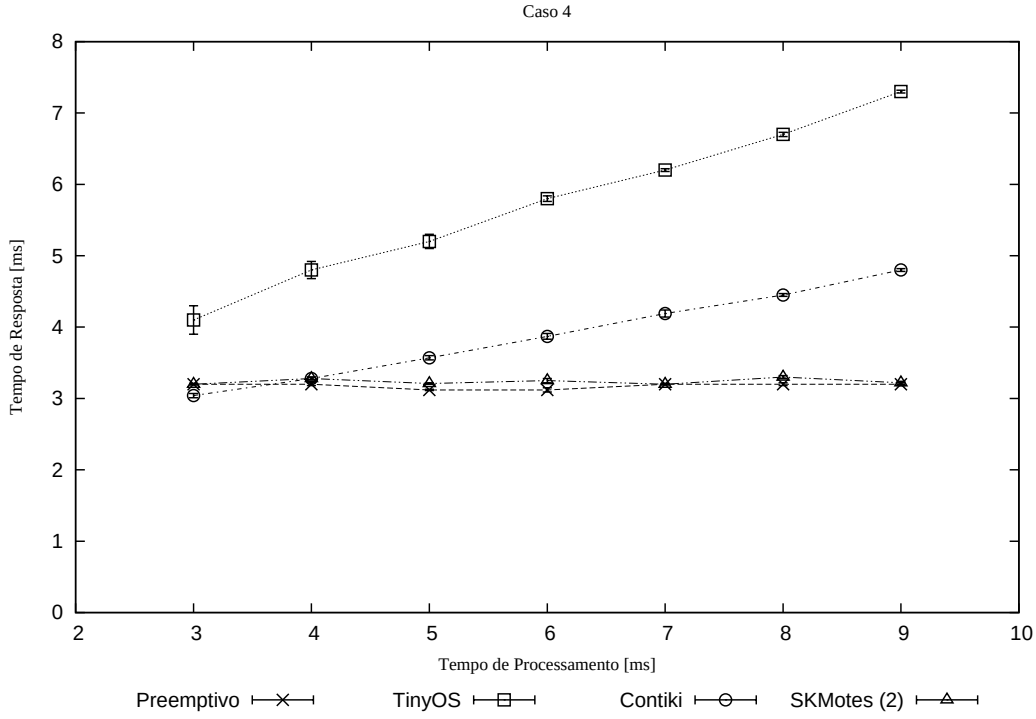

Figura 4.25: tempo de resposta - caso de teste 4.

Os resultados medidos para esse caso de teste são apresentados nos gráficos das Figuras 4.24 e 4.25. A ocupação da CPU foi similar para o SKMotes, kernel preemptivo e o Contiki. Contudo, o TinyOS demonstrou maior utilização do processador. Em relação ao tempo de resposta, o SKMotes e o kernel preemptivo tiveram resultados similares com variação mínima no tempo de resposta. Entretanto, os kernels de eventos sofreram variação proporcional ao tempo de processamento da tarefa com longo período de computação.

4.6.6 Caso de Teste 5

Nos casos de teste anteriores, o comportamento do SKMotes foi bastante similar ao kernel preemptivo, contudo é esperada uma diferença maior no desempenho conforme a carga computacional se eleve. Para verificar esta situação, um novo caso de teste é proposto. Neste caso, a carga computacional é composta pelas aplicações *Blink 1*, *longCompute 1* e *longCompute 2*. Novamente, o tempo de execução T das tarefas *longCompute* varia de 3.0 ms até 9.0 ms, com incrementos de 1.0 ms. As métricas de avaliação foram ocupação da CPU e tempo de resposta da tarefa *Blink 1*. Neste caso de teste, o SKMotes foi configurado para utilizar dois contextos de execução.

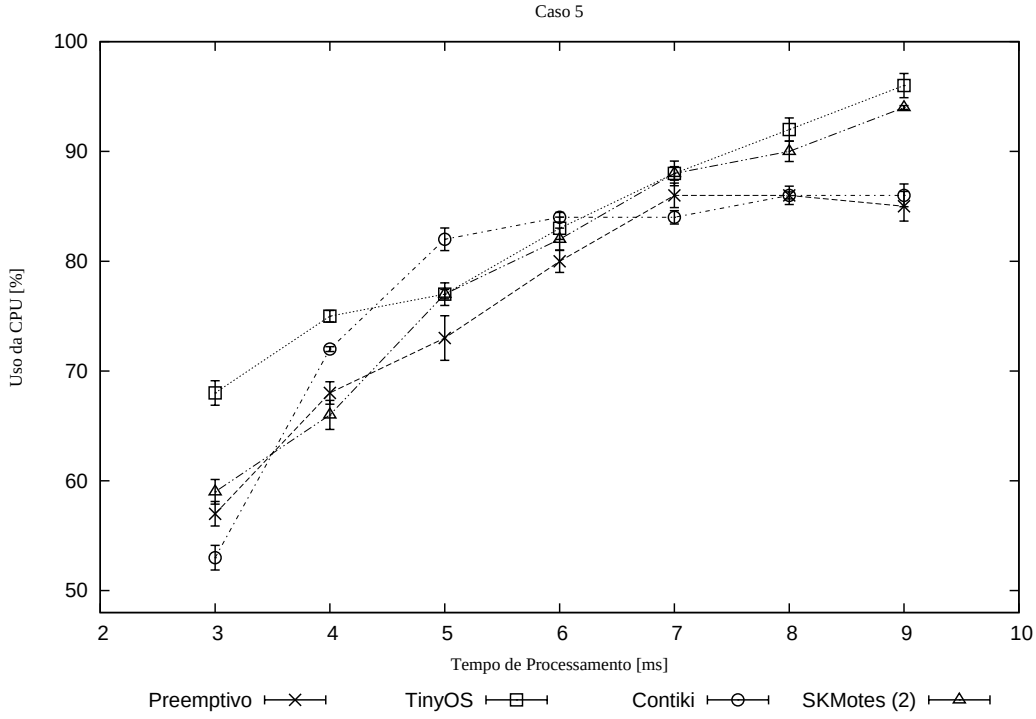

Figura 4.26: uso da CPU - caso de teste 5.

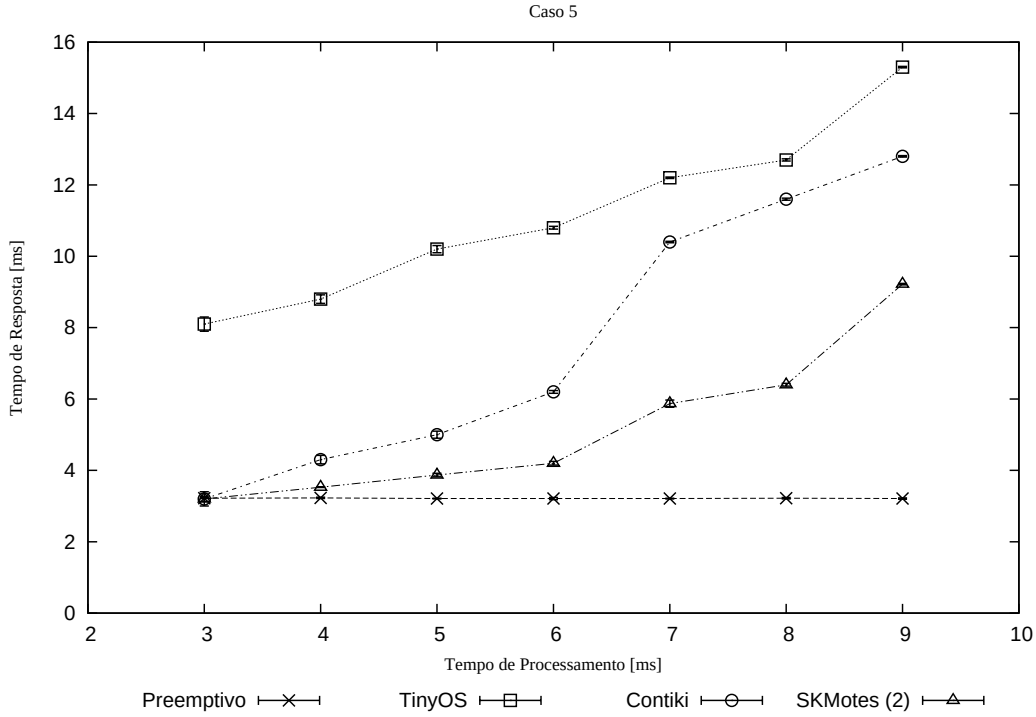

Figura 4.27: tempo de resposta - caso de teste 5.

Os resultados medidos para esse caso de teste são mostrados nos gráficos das Figuras 4.26 e 4.27. A ocupação da CPU é similar para todos os kernels. Em relação ao tempo

de resposta, apenas o kernel preemptivo é capaz de manter o tempo de resposta baixo. As outras soluções sofrem com os atrasos causados pelas *threads* com longo tempo de execução. Entretanto, uma forma de diminuir o impacto de várias tarefas com longo perído de execução no SKMotes é alocar um contexto de execução exclusivo para *threads* com longos períodos de computação.

4.6.7 Alocação de memória.

Além da ocupação da CPU e do tempo médio de resposta, é válido analisar o impacto de cada sistema na ocupação de memória do microcontrolador. A Figura 4.28 apresenta a ocupação absoluta e percentual da memória *Flash* nos quatro primeiros casos de teste. O TinyOS apresenta a maior ocupação da memória, seguido pelo Contiki. A solução preemptiva apresenta o menor consumo de memória. Para estes testes, são utilizados apenas os recursos necessários para execução das tarefas, todo os módulos não utilizados foram removidos dos sistemas.

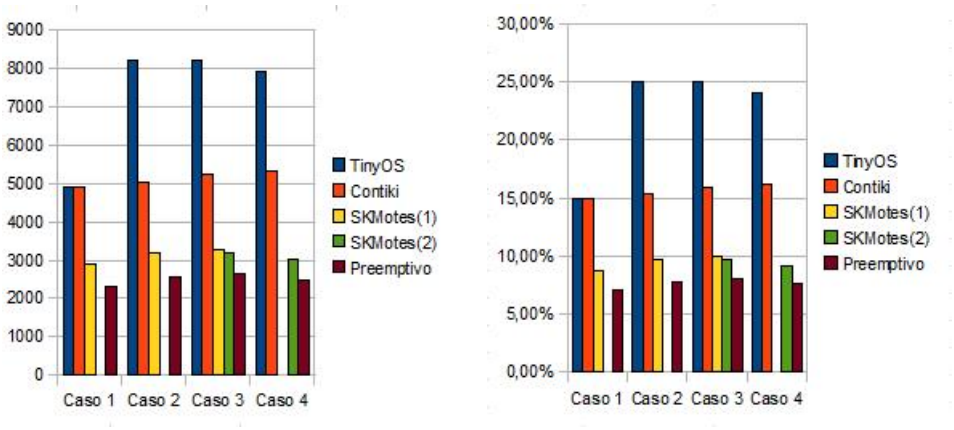

Figura 4.28: alocação de memória *Flash*.

A alocação de RAM, nos quatro últimos casos de teste, é mostrada na Figura 4.29. O TinyOS obtém a menor alocação de memória. Nos quatro cenários, o Contiki apresenta consumo de memória baixo, mas superior ao TinyOS. As soluções preemptiva e o SKMotes apresentam a maior alocação de memória. É importante notar que, nos dois últimos cenários, a solução preemptiva obtém a maior alocação, devido ao fato de que cada *thread* deve possuir um contexto de execução próprio.

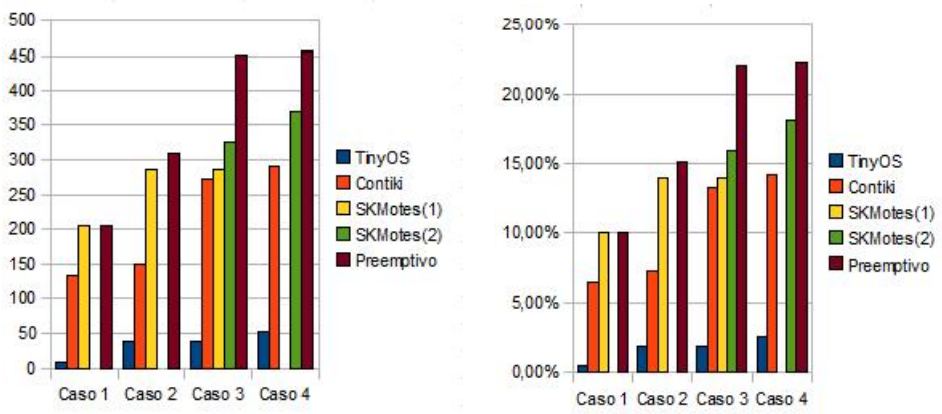

Figura 4.29: alocação de memória RAM.

4.7 Conclusão

O SKMotes é um SO voltado para sistemas computacionais dedicados baseados em microcontroladores simples, que necessitam lidar com concorrências de diferentes tarefas em um ambiente com poucos recursos de memória e poder computacional, semelhantes aos encontrados nos módulos utilizados em RSSFs. Esse SO propõe o emprego de um modelo de concorrência denominado semipreemptivo que impõe o escalonamento preemptivo baseado em prioridades para um sub-conjunto das *threads* do sistema. O restante das *threads* permanece em espera, ocupando apenas um contexto mínimo de execução, que não contempla a pilha de dados. Essa abordagem é alcançada através da utilização de chamadas de sistema especiais que bloqueiam as *threads* que esperam por eventos específicos, de forma que o contexto de execução seja apenas o ponto de continuação na rotina principal da *thread*. O principal objetivo desse modelo é provê tempos de resposta baixos para *threads* de alta prioridade, ao mesmo tempo que garante baixa ocupação de memória, baixo consumo de energia e um modelo de programação baseado em *threads*.

Embora, o modelo proposto possua diversas vantagens, é importante ressaltar algumas de suas limitações. Em primeiro lugar, o programador deve restringir o uso de variáveis locais em suas tarefas, trocando-as por variáveis globais ou estáticas. Essa premissa pode ser garantida com o uso de ferramentas que façam a análise do código antes do processo de compilação. Além disso, as chamadas de sistema de bloqueio só podem ser executadas na rotina principal da *thread*.

Como os contextos de execução são reduzidos em relação ao número de unidades de código executáveis no sistema, é possível que *threads* de alta prioridade permaneçam no estado de espera, caso todos os contextos estejam alocados, causando uma latência

variável e uma situação de inversão de prioridades. Esse problema, pode ser reduzido pela adoção de contextos de execução específicos para *threads* de alta prioridade. Contudo, o programador deve garantir que essas *threads* tenham tempo de processamento curto.

Outro aspecto que requer atenção na utilização desse modelo, é a alocação da pilha de dados. Como as *threads* compartilham diferentes contextos de execução, todos os contextos devem possuir o mesmo tamanho de pilha, que deve ser dimensionada pela demanda de chamadas de rotinas nas *threads*. Um *software* pode fazer a análise estática do código para indicar qual é o tamanho ótimo da pilha, mas de qualquer forma essa abordagem gera um certo desperdício de memória.

A Tabela 4.2 mostra o desempenho médio dos *kernels* de cada modelo de concorrência relativo ao desempenho médio apresentado pelo SKMotes. Nesse caso, as métricas usadas são ocupação da CPU, a ocupação da memória RAM e o atendimento das restrições de tempo das tarefas de acordo com os resultados apresentados nos casos de testes. Nesta tabela, as métricas calculadas de cada *kernel* são divididas pelo desempenho do SKMotes para demonstrar o desempenho relativo a este SO.

	Kernels Baseados em Eventos	*Kernel* Preemptivo	SKMotes
Ocupação da CPU	0,99	0,94	1
Utilização da RAM	0,41	1,20	1
Tempo de resposta	1,34	0,84	1

Tabela 4.2: desempenho médio dos *kernels* relativo ao SKMotes.

Pode-se perceber que o SKMotes apresenta ocupação de memória RAM, em média, 20% inferior ao *kernel* preemptivo, além disso o tempo de resposta do SKMotes é 34% mais baixo do que os apresentados pelos *kernels* baseados em eventos utilizados na avaliação de desempenho.

Na Tabela 4.3, temos o desempenho médio dos *kernels* relativo ao SKMotes, referente apenas aos casos de teste 4 e 5. Esses casos de teste possuem *threads* orientadas a E/S e *threads* orientadas a longos períodos de computação. Para esses cenários, o SKMotes apresenta tempo de resposta, em média, 63% inferior aos apresentados pelos *kernels* baseados em eventos, ao mesmo tempo em que apresenta ocupação de RAM, em média, 23% inferior em relação ao *kernel* preemptivo.

A avaliação de desempenho mostrou, para os casos de teste, que o SKMotes usando o escalonamento semipreemptivo é adequado para sistemas que possuem conjunto de tarefas com demandas voltadas para E/S e para longos períodos de computação, como: compressão de dados; filtro digitais e criptografia. Os resultados dos casos de testes mostram que os *kernels* baseados em eventos não conseguem atender as demandas temporais das

	Kernels Baseados em Eventos	*Kernel* Preemptivo	SKMotes
Ocupação da CPU	1,16	0,95	1
Utilização da RAM	0,46	1,23	1
Tempo de resposta	1,63	0,75	1

Tabela 4.3: desempenho médio dos *kernels* relativo ao SKMotes nos casos de teste 4 e 5.

tarefas em ambientes com tarefas com longos períodos de computação, já o kernel preemptivo atende tais demandas, mas tem alta ocupação de memória, exigindo uma pilha de dados para cada *thread* do sistema.

Por fim, o SKMotes se mostrou um SO com características atraentes para aplicações de RSSFs, que demandem um *kernel* com baixa ocupação de memória e de CPU e com desempenho comparável ao de um *kernel* preemptivo em relação ao tempo de resposta.

Capítulo 5

Conclusões, Contribuições e Trabalhos Futuros

Ne sta dissertação é proposto um novo sistema operacional (SO) para aplicações em RSSFs, chamado SKMotes, que utiliza um modelo de concorrência baseado em *threads*, mas não completamente preemptivo, pois em dado momento apenas um subconjunto das *threads* do sistema estão executando no modo preemptivo baseado em prioridades. O principal objetivo deste modelo é prover tempos de resposta baixos para *threads* de alta prioridade, ao mesmo tempo em que garante consumo de energia equivalente a um *kernel* preemptivo, modelo de programação baseado em *threads*, ocupação de memória de dados menor do que um *kernel* preemptivo e tempos de resposta inferiores aos apresentados por *kernels* baseados em eventos.

Para avaliar o SKMotes, são especificados casos de testes que refletem aspectos reais utilizados por aplicações em RSSFs, tais como: utilização dos periféricos de comunicação e tarefas com requisitos de tempo. A avaliação do desempenho do SKMotes é realizada na plataforma de aplicações embarcadas em rede eZ430-RF2500 (TEXAS INSTRUMENTS, 2011), baseada no microcontrolador MSP430 da Texas Instruments, bem como, em um ambiente de testes projetado utilizando um kit de avaliação baseado em uma FPGA da família Spartan 3E (DIGILENTINC, 2011). Um circuito digital descrito em VHDL foi desenvolvido especificamente para essa aplicação.

Os casos de teste permitem avaliar o desempenho do novo SO em relação a outros sistemas utilizados, em aplicações de RSSFs, como o TinyOS (HILL et al., 2000) e o Contiki (DUNKELS; GRONVALL; VOIGT, 2004), além de uma versão completamente preemptiva do SKMotes.

Os testes de avaliação de desempenho mostram que, para os casos de teste realizados,

o SKMotes apresenta ocupação do processador equivalente às soluções baseadas em *multithreading* preemptivo, mas com consumo de memória de dados, em média, 20% menor. Além disso, o SKMotes é capaz de garantir tempos de respostas, em média, 34% inferiores às soluções baseadas em *kernels* de eventos. Quando se avalia apenas os casos de teste que possuem *threads* orientadas à E/S e a longos períodos de computação, o tempo de resposta chega a ser, em média, 63% inferior ao apresentado por *kernels* baseados em eventos. Os valores percentuais apresentados foram calculados a partir da consolidação dos resultados de todos os casos de testes realizados durante a avaliação de desempenho.

Contudo, o modelo proposto impõe algumas restrições ao modelo de programação. Por exemplo, as variáveis utilizadas na função principal da *thread* devem ter escopo global ou estático. Além disso, os pontos de bloqueio só podem ser executados da função principal, pois o histórico de chamadas de rotinas, que é armazenado na pilha de dados, não é salvo. Também, em algumas situações, as *threads* de alta prioridade podem ter que esperar por um contexto de execução, o que adiciona uma latência variável na execução. Entretanto, estas limitações são compensadas pelos ganhos na ocupação da memória, estilo de programação e consumo de energia.

As principais contribuições desta dissertação são: concepção, implementação e avaliação de um novo SO para RSSFs; e implementação de um ambiente de avaliação de desempenho de sistemas computacionais baseados em microcontroladores. Este novo SO para RSSFs possui características que permitem que este seja empregado em aplicações de RSSFs que executem simultaneamente tarefas orientadas à E/S e a longos períodos de computação. Por exemplo, aplicações de RSSFs que realizem funções de compressão de dados (TAVLI; BAGCI; CEYLAN, 2010), criptografia (WANDER et al., 2005), dentre outras, que demandem longos períodos de computação. Por sua vez, o ambiente de avaliação de desempenho proposto é capaz de realizar medições de diversos parâmetros de desempenho da plataforma alvo, bem como interagir com esta plataforma através da troca de pacotes pela interface de comunicação serial. Este ambiente de avaliação pode ser reutilizado em diferentes cenários de avaliação de desempenho de sistemas computacionais baseados em microcontroladores.

Note que o SKMotes não está ainda em condições de suportar as mais variadas plataformas de *hardware* disponíveis para RSSFs. Assim, os trabalhos futuros podem aprimorá-lo para aumentar suas funcionalidades e aplicações, dentre os quais podem-se citar o porte para plataformas de *hardware* adicionais, bem como a implementação de um módulo que faça a alocação dinâmica dos contextos de execução de acordo com métricas do escalonamento das *threads*. Por exemplo, o sistema, ao detectar um aumento da latência das

threads com restrições de tempo, pode aumentar o número de contextos alocados para esta categoria de *threads*. Além dessas funcionalidades, o SKMotes pode ser melhorado através da implantação de um serviço de atualização do *firmware* do nó sensor durante o seu funcionamento em campo, este serviço pode ser construído a partir de um módulo de ligação dinâmica, que permita a inserção de novos módulos de software na imagem da memória *flash* do microncotrolador.

Referências Bibliográficas

ABBASI, A. A.; YOUNIS, M. A survey on clustering algorithms for wireless sensor networks. *Comput. Commun.*, Butterworth-Heinemann, Newton, MA, USA, v. 30, p. 2826–2841, October 2007. ISSN 0140-3664. Disponível em: <http://dx.doi.org/10.1016/j.comcom.2007.05.024>.

ABRACH, H. et al. Mantis: system support for multimodal networks of in-situ sensors. In: *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. New York, NY, USA: ACM, 2003. (WSNA '03), p. 50–59. ISBN 1-58113-764-8. Disponível em: <http://doi.acm.org/10.1145/941350.941358>.

BECKWITH, R.; TEIBEL, D.; BOWEN, P. Pervasive computing and proactive agriculture. In: *Proceedings of the 2nd International Conference on Pervasive Computing*. Vienna, Austria: ACM, 2004. (Pervasive '04).

BHATTI, S. et al. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 10, p. 563–579, August 2005. ISSN 1383-469X. Disponível em: <http://dx.doi.org/10.1145/1160162.1160178>.

BOUKERCHE, A. et al. Localization in time and space for wireless sensor networks: A mobile beacon approach. *A World of Wireless, Mobile and Multimedia Networks, International Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1–8, 2008.

BREZA, M.; MCCAN, J. A. Lessons in implementing bio-inspired algorithms on wireless sensor networks. In: ESA CONFERENCE ON ADAPTIVE HARDWARE AND SYSTEMS. [S.l.], 2008.

CAMILO, T. et al. An Energy-Efficient Ant-Based Routing Algorithm for Wireless Sensor Networks. *Ant Colony Optimization and Swarm Intelligence*, p. 49–59, 2006. Disponível em: <http://dx.doi.org/10.1007/11839088_5>.

CHATTERJEA, S. et al. A distributed and self-organizing scheduling algorithm for energy-efficient data aggregation in wireless sensor networks. *ACM Transactions on Sensor Networks*, v. 4, 2008.

CHEHRI, A.; FORTIER, P.; TARDIF, P. M. Uwb-based sensor networks for localization in mining environments. *Ad Hoc Networks*, n. 7, p. 987–1000, 2009.

CHRISTIN, D.; MOGRE, P. S.; HOLLICK, M. Survey on wireless sensor network technologies for industrial automation: The security and quality of service perspectives. *Future Internet*, v. 2, n. 2, p. 96–125, 2010.

CULLER, D.; ESTRIN, D.; SRIVASTAVA, M. Overview of sensors network. *IEEE Computer Magazine*, v. 37, p. 41–49, 2004.

DIGILENTINC. *Kit FPGA Spartan 3E*. 2011. Disponível em: <http://www.digilentinc.com/Data/Products/S3EBOARD/S3EStarter_ug230.pdf>.

DONG, W. et al. Providing os support for wireless sensor networks: Challenges and approaches. *Communications Surveys Tutorials, IEEE*, v. 12, n. 4, p. 519 –530, 2010. ISSN 1553-877X.

DUNKELS, A.; GRONVALL, B.; VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In: *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. Washington, DC, USA: IEEE Computer Society, 2004. (LCN '04), p. 455–462. ISBN 0-7695-2260-2. Disponível em: <http://dx.doi.org/10.1109/LCN.2004.38>.

DUNKELS, A. et al. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2006. (SenSys '06), p. 29–42. ISBN 1-59593-343-3. Disponível em: <http://doi.acm.org/10.1145/1182807.1182811>.

GABER, M. M.; RöHM, U.; HERINK, K. An analytical study of central and in-network data processing for wireless sensor networks. *Inf. Process. Lett.*, p. 62–70, 2009.

GAO, X. et al. A multi- hop energy efficient clustering algorithm in routing protocol for wireless sensor networks. In: *Proceedings of the 5th International Conference on Wireless communications, networking and mobile computing*. Piscataway, NJ, USA: IEEE Press, 2009. (WiCOM'09), p. 3087–3091. ISBN 978-1-4244-3692-7. Disponível em: <http://dl.acm.org/citation.cfm?id=1737966.1738220>.

GAY, D. et al. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 38, p. 1–11, May 2003. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/780822.781133>.

GUI, C.; MOHAPTRA, P. Power conservation and quality of surveillance on target tracking sensor networks. In: *10th annual international conference on Mobile computing and networking*. [S.l.: s.n.], 2004.

GUO, S. et al. Opportunistic flooding in low-duty-cycle wireless sensor networks with unreliable links. In: *MOBICOM*. [S.l.: s.n.], 2009. p. 133–144.

HANDE, A. et al. Indoor solar energy harvesting for sensor network router nodes. *Microprocessors and Microsystems*, v. 31, n. 6, p. 420–432, 2007.

HEINZELMAN, W. R.; CHANDRAKASAN, A.; BALAKRISHNAN, H. Energy-efficient communication protocol for wireless microsensor networks. In: *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8 - Volume 8*. Washington, DC, USA: IEEE Computer Society, 2000. (HICSS '00), p. 8020–. ISBN 0-7695-0493-0. Disponível em: <http://portal.acm.org/citation.cfm?id=820264.820485>.

HEINZELMAN, W. R.; KULIK, J.; BALAKRISHNAN, H. Adaptive protocols for information dissemination in wireless sensor networks. In: *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. New York, NY, USA: ACM, 1999. (MobiCom '99), p. 174–185. ISBN 1-58113-142-9. Disponível em: <http://doi.acm.org/10.1145/313451.313529>.

HILL, J. et al. System architecture directions for networked sensors. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, p. 93–104, November 2000. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/356989.356998>.

INTANAGONWIWAT, C.; GOVINDAN, R.; ESTRIN, D. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In: *6th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. [S.l.: s.n.], 2000.

JÚNIOR, O. A. de L.; CASTRO, H. S.; CORTEZ, P. C. Um algoritmo emergente para coleta de dados em redes de sensores sem fio. In: *Workshop de Sistemas Embarcados*. Gramado: [s.n.], 2010. p. 109–117.

JUANG, P. et al. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. *ASPLOS-X 2002: Proceedings of the 10th*

international conference on Architectural support for programming languages and operating systems, ACM, New York, NY, USA, v. 37, n. 10, p. 96–107, out. 2002. ISSN 0362-1340. Disponível em: <http://dx.doi.org/10.1145/605397.605408>.

KASTEN, O.; RöMER, K. Beyond event handlers: programming wireless sensors with attributed state machines. In: *Proceedings of the 4th international symposium on Information processing in sensor networks*. Piscataway, NJ, USA: IEEE Press, 2005. (IPSN '05). ISBN 0-7803-9202-7. Disponível em: <http://portal.acm.org/citation.cfm?id=1147685.1147695>.

KEMPE, D.; DOBRA, A.; GEHRKE, J. Gossip-based computation of aggregate information. In: . [S.l.]: 44th Annual IEEE Symposium on Foundations of Computer Science, 2003.

KLUES, K. et al. Tosthreads: thread-safe and non-invasive preemption in tinyos. In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. New York, NY, USA: ACM, 2009. (SenSys '09), p. 127–140. ISBN 978-1-60558-519-2. Disponível em: <http://doi.acm.org/10.1145/1644038.1644052>.

LAUKKARINEN, T. et al. Hybridkernel: Preemptive kernel with event-driven extension for resource constrained wireless sensor networks. In: *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*. [S.l.: s.n.], 2009. p. 161 –166. ISSN 1520-6130.

LIN, F.-S.; YEN, H.-H.; LIN, S.-P. A Novel Energy-Efficient MAC Aware Data Aggregation Routing in Wireless Sensor Networks. *Sensors 2009*, ACM, v. 9, 2011.

LINDSEY, S.; RAGHAVENDRA, C.; SIVALINGAM, K. Data gathering in sensor networks using the energy*delay metric. In: *Parallel and Distributed Processing Symposium., Proceedings 15th International*. [s.n.], 2001. p. 2001–2008. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=925196>.

PEGASIS: Power-efficient gathering in sensor information systems, v. 3. 3-1125–3-1130 vol.3 p. Disponível em: <http://dx.doi.org/10.1109/AERO.2002.1035242>.

MAINWARING, A. et al. Wireless sensor networks for habitat monitoring. In: *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. New York, NY, USA: ACM, 2002. (WSNA '02), p. 88–97. ISBN 1-58113-589-0. Disponível em: <http://doi.acm.org/10.1145/570738.570751>.

MCCARTNEY, W. P.; SRIDHAR, N. Abstractions for safe concurrent programming in networked embedded systems. In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2006. (SenSys '06), p. 167–180. ISBN 1-59593-343-3. Disponível em: <http://doi.acm.org/10.1145/1182807.1182825>.

MOTTOLA, L.; PICCO, G. P. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 43, abr. 2011. ISSN 0360-0300. Disponível em: <http://dx.doi.org/10.1145/1922649.1922656>.

SARUWATARI, S.; SUZUKI, M.; MORIKAWA, H. A compact hard real-time operating system for wireless sensor nodes. In: *Proceedings of the 6th international conference on Networked sensing systems*. Piscataway, NJ, USA: IEEE Press, 2009. (INSS'09), p. 167–174. ISBN 978-1-4244-6313-8. Disponível em: <http://portal.acm.org/citation.cfm?id=1802340.1802381>.

SEAH, W. K. G.; EU, Z. A.; TAN, H. Wireless sensor networks powered by ambient energy harvesting (wsn-heap) - survey and challenges. *2009 1st International Conference on Wireless Communication Vehicular Technology Information Theory and Aerospace Electronic Systems Technology*, Ieee, p. 1–5, 2009. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5172411>.

SINGH, S. K.; SINGH, M. P.; SINGH, D. K. Routing protocols in wireless sensor networks - a survey. *International Journal of Computer Science & Engineering (IJCSES)*, v. 1, n. 2, 2010.

TAVLI, B.; BAGCI, I. E.; CEYLAN, O. Optimal data compression and forwarding in wireless sensor networks. *Comm. Letters.*, IEEE Press, Piscataway, NJ, USA, v. 14, p. 408–410, May 2010. ISSN 1089-7798. Disponível em: <http://dx.doi.org/10.1109/LCOMM.2010.05.092372>.

TEXAS INSTRUMENTS. *Módulo sem fio ez430-RF2500*. 2011. Disponível em: <http://focus.tij.co.jp/jp/lit/ug/slau227e/slau227e.pdf>.

TILAK, S.; ABU-GHAZALEH, N. B.; HEINZELMAN, W. A taxonomy of wireless micro-sensor network models. *SIGMOBILE Mob. Comput. Commun. Rev.*, ACM Press, New York, NY, USA, v. 6, n. 2, p. 28–36, abr. 2002. ISSN 1559-1662. Disponível em: <http://dx.doi.org/10.1145/565702.565708>.

TRUMPLER, E.; HAN, R. A systematic framework for evolving tinyos. In: *In proc. 3 rd Workshop on Embedded Networked Sensors*. [S.l.: s.n.], 2006. p. 61–65.

VIDHYAPRIYA, R.; VANATHI, P. T. Energy efficient data compression in wireless sensor networks. *Int. Arab J. Inf. Technol.*, v. 6, n. 3, p. 297–303, 2009.

WANDER, A. et al. Energy analysis of public-key cryptography for wireless sensor networks. In: *PerCom*. [S.l.]: IEEE Computer Society, 2005. p. 324–328. ISBN 0-7695-2299-8.

WERNER-ALLEN, G. et al. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 10, p. 18–25, March 2006. ISSN 1089-7801. Disponível em: <http://dx.doi.org/10.1109/MIC.2006.26>.

WILLIAMS, J. W.; BERGMANN, N. Topology optimization in wireless sensor networks for precision agriculture applications. In: *SensorComm 2007. International Conference on Sensor Technologies and Applications*. Valência, Espanha: [s.n.], 2007. p. 526–530. ISBN 978-0-7695-2988-2.

YICK, J.; MUKHERJEE, B.; GHOSAL, D. Wireless sensor network survey. *Comput. Netw.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 52, p. 2292–2330, August 2008. ISSN 1389-1286. Disponível em: <http://portal.acm.org/citation.cfm?id=1389582.1389832>.

ZOU, Y.; CHAKRABARTY, K. Redundancy analysis and a distributed self-organization protocol for fault-tolerant wireless sensor networks. *IJDSN*, v. 3, n. 3, p. 243–272, 2007.